FINCoS++: Marco de análisis de rendimiento de motores para procesamiento de eventos complejos

FINCoS++: A framework for performance analysis of complex event processing engines

Trabajo de Fin de Máster

Curso 2019 - 2020

Autor
John Alexander Torres Rivera

Director
Jesús Correas Fernández
María de las Mercedes García Merayo

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

FINCoS++: Marco de análisis de rendimiento de motores para procesamiento de eventos complejos

FINCoS++: A framework for performance analysis of complex event processing engines

Trabajo de Fin de Máster en Ingeniería Informática

Autor
John Alexander Torres Rivera

Director
Jesús Correas Fernández
María de las Mercedes García Merayo

Convocatoria: Septiembre de 2020
Calificación: 9.0

Máster en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

7 de septiembre de 2020

Resumen

El procesamiento de eventos complejos (CEP) se ha convertido en uno de los campos que han emergido más rápidamente como disciplina de investigación y tendencia industrial. La habilidad de recopilar, analizar, y reaccionar a los eventos en tiempo real se ha vuelto un componente clave de los sistemas de información. Si embargo, aún existe una carencia de información sobre el rendimiento de los motores de procesamiento de eventos complejos.

FINCoS es una herramienta que cuenta con un enfoque flexible y neutro a través del cual los usuarios pueden ejecutar rápidamente pruebas de rendimiento en una plataforma de procesamiento de eventos sin tener que configurar las rutinas de generación de carga y la conversión de eventos. A pesar de ello esta herramienta se ha quedado desactualizada con la última versión publicada en 2013, lo que hace que no se pueda utilizar con los motores CEP actuales en un área tan dinámica como es la del procesamiento de eventos complejos. Como solución a este problema en este Trabajo de Fin de Máster se ha desarrollado *FINCoS++*.

Las principales diferencias entre FINCoS y FINCoS++ son las siguientes: se ha actualizado y extendido el adaptador con motores CEP y se ha integrado un motor para procesamiento de eventos nuevo, nuevas funciones automatizadas para la generación de esquemas y patrones, la posibilidad de generar pruebas de rendimiento de distintos motores CEP en una o diferentes máquinas simultáneamente, y el acceso a la visualización de las estadísticas recogidas dentro de la misma aplicación.

El presente trabajo explica las definiciones esenciales para entender el procesamiento de eventos complejos, analiza los principales trabajos relacionados con comparación de rendimiento de estos motores y el progreso que ha tenido este campo, describe el desarrollo de FINCoS++ y su arquitectura, las mejoras incluidas en esta nueva versión, así como los pasos realizados para llegar a dichas mejoras. También se muestran las pruebas y resultados realizados en FINCoS++ para ilustrar su funcionamiento, concluyendo con algunas posibles ampliaciones al sistema.

Palabras clave

Evento simple, Evento complejo, Procesamiento de eventos complejos, Motor CEP, Stream, Source, Sink, Patrón de eventos

Abstract

Complex event processing (CEP) has become one of the fastest-growing emerging fields as a research area and industrial trend. The ability to collect, analyze, and react to events in real time has become a key component of information systems. However, there is still a lack of information on the performance of complex event processing engines.

FINCoS is a framework that has flexible and neutral approach through which users can quickly run realistic performance tests on an event processing platform without having to configure the load generation routines and event conversion. Despite this, the FINCoS framework has not been updated since 2013, and therefore it cannot be used with current CEP engines in such a dynamic research area as complex event processing. As a solution to this problem, FINCoS++ has been developed in this Master's Thesis.

The main differences between FINCoS and FINCoS++ are the following: The adapter to CEP engines of FINCoS has been updated and extended for enabling additional CEP engines, and a new event processing engine has been integrated; new automated functions for schema and pattern generation have been added to this framework; it has been enabled the generation of performance tests over different CEP engines on one or more servers simultaneously; and the visualization of the collected statistics has been enabled from the main application.

This thesis explains the essential definitions to understand of complex events processing, analyzes the main works related to the performance comparison of these engines and the progress has been made in this field, describes the development and architecture of FINCoS++, the improvements included in this new version, and the steps taken to reach these improvements. Tests and results carried out in FINCoS++ have been included to illustrate its operation, concluding with some possible extensions to this system.

Keywords

Single event, Complex event, Complex event processing, CEP engine, Stream, Source, Sink, Event pattern

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

Agradecimientos

A mis padres, Jairo Torres y Mireya Rivera por su apoyo incondicional durante toda mi vida y en cada proyecto que emprendo. No solo desarrollar de este trabajo, sino culminar esta Maestría con éxito ha sido un reto, debido a los diferentes obstáculos presentados al momento de cambiar el país de residencia.

A mi hermano y mi cuñada por permitirme habitar en su vivienda durante los momentos de crisis económica y durante la pandemia.

A mis directores de tesis, Jesús y Mercedes por su guía, soporte y entusiasmo durante el desarrollo de este proyecto.

A mis compañeros y amigos de la UCM con los que nos ayudamos mutuamente durante esta empresa: Gonzalo, Javier, Aitor, Juan, y en especial a Santiago.

A todos los amigos, profesores, entrenadores, ex-compañeros de trabajo y familiares que siempre me dieron un aliento para seguir adelante con esta aventura.

CAPÍTULO 1

INTRODUCCIÓN

> "Hacer que lo simple sea complicado es trivial; hacer que lo complicado sea simple, asombrosamente simple, eso es creatividad."
>
> *Charles Mingus*

1.1. Motivación

En el mundo de hoy, una cantidad exorbitante de datos proviene de diferentes fuentes como sensores, transacciones, actividad web o las redes sociales, entre otros. La habilidad para acumular, analizar, y reaccionar al flujo de eventos en tiempo real se está convirtiendo en un componente clave de los sistemas de información, tanto en los negocios como en el campo de la investigación. Dar sentido a este desborde de datos requiere un procesamiento rápido y eficiente.

El procesamiento de un gran volumen de eventos que se transforman en información valiosa es una parte vital de la toma de decisiones, y el procesamiento de eventos complejos se ha convertido en uno de los campos que han surgido más rápidamente como disciplina de investigación y como tendencia industrial.

Sin embargo, todavía carecemos de información acerca del rendimiento de los motores de procesamiento de eventos complejos, ya que hasta la aparición de FINCoS no se disponía de puntos de referencia estándar. El objetivo de FINCoS era llenar este vacío, proporcionando un enfoque flexible y neutral a través del cual los usuarios pudieran ejecutar rápidamente pruebas de rendimiento realistas en uno o más motores de procesamiento de eventos sin tener que configurar personalmente la conversión de eventos y las fases de carga [36].

EL desarrollo de FINCoS comenzó en 2007, como parte del proyecto BiCEP [11]. En ese momento, había muy poca información sobre el uso de sistemas de procesamiento de eventos complejos y una gran diversidad de productos, cada uno con sus propios lenguajes y estilos, lo que planteaba desafíos importantes para el desarrollo de nuevos marcos de análisis. La primer versión de FINCoS fue lanzada en 2008, y su última actualización es de 2013. Desde entonces no se han publicado nuevas actualizaciones sobre FINCoS y la versión que se encuentra disponible cuenta con adaptador para un solo motor, Esper.

Para continuar con este trabajo nos proponemos generar o mejorar una herramienta que cumpla con los objetivos iniciales de FINCoS y que además permita trabajar con diferentes motores de procesamiento de eventos; para llevar a cabo esta propuesta es preciso retomar FINCoS, actualizar su versión, realizar mejoras en su rendimiento, optimizar su funcionalidad y extenderlo a más motores de procesamiento de eventos complejos para realizar pruebas de rendimiento simultáneo.

El reto planteado supone analizar un sistema de la complejidad de FINCoS sin disponer de más material que su código fuente para identificar su funcionamiento a fondo y enfrentarse a las tecnologías que emplea mientras se desarrolla la nueva versión.

1.2. Objetivos

Este trabajo tiene como objetivo general la ampliación y mejora de una herramienta para la generación de carga y para medición del rendimiento de motores CEP de manera flexible y neutra. Esta herramienta proporcionará a los usuarios un sistema que utilice las últimas

versiones de los motores CEP y pueda generar cargas de trabajo artificiales así como utilizar conjuntos de datos reales, permitiendo evaluar diferentes soluciones candidatas. A su vez, esta herramienta se podrá utilizar para que los desarrolladores de motores para procesamiento de eventos complejos puedan comparar y mejorar el rendimiento de los mismos.

Los objetivos específicos de este trabajo son los siguientes:

- Actualizar FINCoS para poder utilizar la última versión de Esper, 8.4. Para ello, se deben identificar las diferencias con la versión 4.9 y modificar los componentes de FINCoS que realizan la comunicación bidireccional con Esper.
- Estudiar diferentes motores CEP para extender el sistema FINCoS con otro de los motores más utilizados.
- Realizar otras mejoras funcionales que permitan mantener la coherencia de los datos de modelos de eventos complejos configurados para FINCoS.
- Evaluar el nuevo sistema con algunos casos de estudio.

1.3. Plan de trabajo

Para lograr el cumplimiento del objetivo principal, es necesario realizar una gestión efectiva del proyecto y dividirlo en hitos funcionales con sus respectivas tareas. La siguiente lista enumera los principales hitos y tareas a alcanzar a lo largo de este trabajo:

1. Familiarización con FINCoS
   a) Revisar documentación disponible.
   b) Instalación y pruebas sobre la versión actual.
   c) Estudio en profundidad del código fuente.
2. Migración a Esper 8.4.
   a) Revisar documentación de Esper.

   b) Analizar los componentes de FINCoS que realizan la comunicación con Esper y modificarlos para adaptarlos a la versión 8.4.

   c) Probar configuración nueva con Esper 8.4.

3. Integración de nuevos motores.

   a) Revisar documentación de motores CEP adicionales.

   b) Seleccionar motores CEP a integrar.

   c) Realizar integración de nuevos motores al sistema.

   d) Probar el funcionamiento de la integración.

4. Realizar mejoras funcionales.

   a) Automatizar el sistema.

   b) Crear nuevos menús.

   c) Integrar informe de estadísticas.

   d) Ajustar definiciones.

5. Pruebas de comparación con los motores integrados.

1.4. Estructura del documento

La estructura de este documento se muestra a continuación:

- En el capítulo 2 se encuentran las definiciones esenciales para entender el trabajo desarrollado acerca del procesamiento de eventos complejos.
- En el capítulo 3 se revisan los principales trabajos relacionados con comparación de rendimiento de estos motores y hasta dónde se ha avanzado en este campo.

- En el capítulo 4 se describe la arquitectura de FINCoS++, su diagrama de clases, las técnicas de programación utilizadas para optimizar su funcionamiento, y las mejoras incluidas en esta nueva versión, así como los pasos ejecutados para llegar a dichas mejoras.
- En el capítulo 5 se presentan las pruebas y resultados realizados en FINCoS++ para ilustrar su funcionamiento.
- En el capítulo 6 se incluye el manual de usuario del sistema y sus requerimientos de funcionamiento.
- En el capítulo 7 se muestran las conclusiones y posibles ampliaciones futuras sobre FINCoS++.
- En el apéndice A se encuentra el diagrama de clases de FINCoS++

CAPÍTULO 2

CONCEPTOS PRELIMINARES

En este capítulo se encuentran las definiciones esenciales para entender el trabajo desarrollado acerca del análisis de rendimiento de motores para procesamiento de eventos complejos.

2.1. Definiciones

Intuitivamente, un motor de procesamiento de eventos complejos recibe *streams* de eventos simples de diversas fuentes o proveedores, analiza los eventos en busca de *patrones* que relacionan los eventos recibidos, y genera *eventos complejos* que notifican los comportamientos detectados. El motor de procesamiento de eventos complejos actúa como un intermediario que proporciona servicios a la capa de aplicación, ocultando la capa de producción donde las fuentes generan una gran cantidad de eventos de bajo nivel. La Figura 2.1 muestra este esquema global.

2.1.1. Procesamiento de Eventos Complejos

El *Gartner's IT Glossary* define el Procesamiento de Eventos Complejos, conocido por sus siglas en inglés como *CEP* (Complex Event Processing), de la siguiente manera:

"El Procesamiento de Eventos Complejos (CEP) es un tipo de computación en la

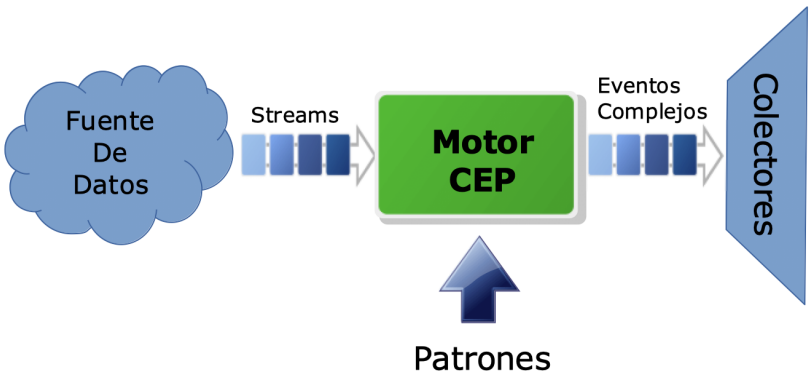


Figura 2.1: Esquema global del Procesamiento de Eventos Complejos

cual los datos sobre eventos entrantes se transforman en datos de eventos complejos más útiles y de mayor nivel, los cuales proporcionan información sobre lo que está sucediendo. La técnica CEP está dirigida por eventos (*eventdriven*), pues el cálculo se activa al recibir datos de eventos. CEP se utiliza para aplicaciones de inteligencia continua altamente exigentes que mejoran el conocimiento de situaciones y respaldan la toma de decisiones en tiempo real." [27]

A pesar de ser un modelo de procesamiento de eventos continuos centrado en detectar relaciones entre eventos, de forma similar a los clásicos *data stream management systems* (DSMS), trata con una clase de consultas sustancialmente diferentes [32].

Los sistemas CEP no solo transforman información en eventos entre el proveedor y el consumidor, sino que también soportan la detección de relaciones entre ellos; por ejemplo, relaciones temporales que pueden ser especificadas por definición o mediante reglas de correlación. A través de la agregación (crear eventos nuevos a partir de uno o más existentes) y la composición (expresar todo en términos de uniones, intersecciones o complementos de otros eventos), se pueden generar nuevos eventos y usarlos subsecuentemente para derivar más eventos abstractos [4].

CEP también es definido como una tecnología para el análisis de streams de eventos

enviados desde múltiples fuentes, para extraer información de alto nivel como forma de detectar los eventos complejos que interesan a las aplicaciones.

Los eventos complejos en los sistemas de información actuales se obtienen a partir de muchos eventos simples, que se pueden observar a través de sensores o mensajes dentro del sistema. Estos eventos observados se denominan *eventos primitivos*.

Los nodos de la red que producen estos eventos se denominan *sources*. La recopilación de información de alto nivel sobre el estado del sistema, o las detección de situaciones de interés en el sistema requieren el análisis de grandes volúmenes de eventos primitivos. En el análisis, los eventos primitivos se correlacionan para detectar *patrones de eventos*. Un patrón de eventos se especifica mediante reglas de correlación de eventos simples. Una regla define un patrón de eventos, el tipo de evento complejo de salida y cómo se deben calcular los datos del evento de salida. Cuando se detectan estos patrones se notifican a los sistemas como *eventos complejos*.

Los sistemas que reciben los eventos complejos detectados se denominan *sinks* [39]. Los sinks se encuentran en la capa de aplicación y son los que interactúan con la misma para mostrar los eventos transformados al usuario.

Un sistema para procesamiento de eventos puede ser dividido en los siguientes componentes funcionales: Especificación de lenguaje, representación de eventos, modelo de procesamiento e interfaz programática.

2.1.2. Lenguaje de Procesamiento de Eventos

Event Processing Language (EPL) es un lenguaje declarativo de tipo *Structured Query Language* (SQL), que utiliza sentencias y cláusulas similares, como `SELECT, FROM, WHERE, GROUP BY, HAVING y ORDER BY`. Los streams reemplazan a las tablas como fuente de datos, y contienen eventos que sustituyen a las filas de las mismas como unidad básica de datos.

Dado que los eventos se componen de datos, también se pueden aplicar de manera efectiva los conceptos procedentes de SQL de correlación de datos a través de sentencias *join*, filtrado

como subconsultas y agregación mediante agrupaciones. La cláusula `INSERT INTO` permite definir nuevos eventos a partir de los correspondientes a diferentes streams para su posterior procesamiento. También están disponibles cláusulas adicionales como `RETAIN, MATCHING` o `OUTPUT` para controlar el procesamiento de los eventos [42].

EPL es usado en motores como Oracle [42], Esper [24] o ESA [33].

2.1.3. Representación de eventos

Los eventos se pueden representar de diferentes maneras según el motor de procesamiento. Algunos tipos de eventos comunes de los motores basados en Java son:

Plain Old Java Object (POJO) Events: Los eventos POJO son instancias de objetos que exponen las propiedades del evento a través de métodos `getter` de estilo JavaBeans. Las clases de eventos o interfaces no tienen que ser totalmente compatibles con la especificación JavaBeans; sin embargo, para que el motor CEP obtenga las propiedades del evento, deben estar presentes los métodos `getter` de JavaBeans requeridos.

Map Events (Util Map): Los eventos Map implementan la interfaz `com.bea.wlevs.ede.api.MapEventObject`, la cual extiende la interfaz `java.util.Map`. Las propiedades de los eventos tipo Map son los valores de cada entrada accesible a través del método `get` expuesto por la interfaz `java.util.Map`.

Object Array: Los eventos object-array son matrices de objetos donde cada elemento de la matriz es una propiedad.

JSON: Los eventos JSON son documentos con formato JSON de tipo *String*. El documento JSON analizado implementa la interfaz `JsonEventUnderlying` [24, 42].

2.1.4. Patrones de Eventos

Cada evento contiene información que debe ser relacionada con la observada en otros eventos para que dicha información adquiera sentido lógico. Estas relaciones se plasman mediante patrones de eventos.

Los patrones de eventos tratan con diferentes tipos de eventos que son analizados en conjunto, lo cual permite detectar situaciones de interés. Un patrón puede tratar con miles de eventos en un corto periodo de tiempo, lo que ayuda a detectar situaciones relevantes que permiten tomar decisiones de forma rápida [10].

Así pues, un patrón de eventos describe la estructura de un evento (simple o complejo) y las condiciones para su detección. Teniendo esto presente, se pueden clasificar los patrones de acuerdo a su nivel de éxito (buenas y malas soluciones), su nivel de abstracción, su objetivo previsto y su nivel de administración (táctico, estratégico u operacional) [3].

Considerando que la selección del patrón puede definir el éxito del análisis en el procesamiento de los eventos, surge una definición más general y a su vez más extensa del término:

> "Cada patrón es una regla de 3 partes, la cual expresa una relación entre cierto contexto, un cierto sistema de fuerzas que ocurren repetidamente en dicho contexto, y una cierta configuración de software la cual permite que esas fuerzas se resuelvan por si mismas." [5, 21]

2.1.5. Streams

Un stream es una secuencia de eventos ordenados en el tiempo, con un conjunto prefijado de atributos que definen su esquema. Se trata de una secuencia *append-only*, es decir que no se pueden eliminar eventos, solo agregar.

Los streams son los componentes básicos del procesamiento de eventos [24, 49].

2.1.6. Sources

Las sources generan eventos a través de múltiples dispositivos: sensores, computadoras, almacenes de datos u otros; a menudo se generan en tiempo real y en varios formatos de datos, y los envían a streams para su procesamiento [37].

La configuración de la fuente de datos permite definir un mapeo para convertir cada evento entrante de su formato de datos nativo a un evento soportado por el motor CEP específico.

Algunos ejemplos de formatos soportados por determinados motores son: HTTP, Kafka, TCP, In-memory, WSO2Event (específico para Siddhi), Email, JMS, archivos de texto, RabbitMQ, MQTT, WebSocket, Twitter, Amazon SQS, CDC, Prometheus y CSV [24, 49].

2.1.7. Sinks

Los sinks consumen streams de eventos previamente procesados y los envían por diferentes medios a *endpoints* externos en varios formatos de datos.

El sink proporciona una forma de publicar eventos de un stream a sistemas externos mediante la conversión del evento en un formato compatible [49].

La configuración del sink permite a los usuarios definir un mapeo para convertir los eventos al formato de datos de salida requerido (como JSON, TEXT, XML, etc.) y publicar los eventos en los endpoints configurados. Cuando no se proporcionan personalizaciones para tales asignaciones, el motor convierte los eventos al formato de evento predefinido según la definición del stream y el tipo de asignador configurado, antes de publicar los eventos.

2.2. Motores para Procesamiento de Eventos

Un marco integral que incorpore los componentes clave de un sistema de procesamiento de eventos complejo ideal consta de cuatro motores centrales.

- Motor CEP orientado a la computación
- Motor CEP orientado a la detección
- Motor CEP orientado a la predicción
- Motor de reglas

Además de estos motores, se requiere un procesador de consultas, interfaces de consultas para admitir la interacción con diversas bases de datos, así como adaptadores de entrada y salida para actuar como puertas de enlace a los sistemas CEP [44].

Para entender mejor el funcionamiento específico de los motores CEP y poder compararlos se describen a continuación algunos de los más relevantes.

2.2.1. Esper

Esper es un lenguaje, compilador y runtime para CEP y *streaming analytics*, disponible tanto para Java como para .NET. Esper utiliza EPL que implementa y amplía el standard SQL-92 y permite expresiones sobre eventos y tiempo.

Esper incluye un compilador de consultas EPL que genera código de bytes de Java y se pueden guardar en un paquete de archivos con formato `.jar` para su distribución y ejecución.

El *runtime* de Esper carga y ejecuta el código de bytes producido por el compilador de Esper. El objetivo del runtime es proporcionar un motor de procesamiento en memoria altamente escalable, eficiente en memoria, latencia mínima, con capacidad de transmisión para datos en tiempo real y de procesamiento de bancos de datos de diferentes tipos. Tiene una arquitectura de escalamiento horizontal basada en Apache Kafka y Apache Zookeeper, es similar a la de otros lenguajes de programación que se compilan en bytecode JVM, como Scala, Clojure y Kotlin.

Las prioridades del diseño de Esper son:

- Conseguir baja latencia y alto rendimiento.
- Tener expresividad, concisión, extensibilidad del lenguaje EPL.
- Cumplimiento de estándares y mejores prácticas.
- Ligereza en términos de uso de memoria, CPU y E/S.

Dentro de su configuración utiliza librerías externas:

- El compilador y el runtime requieren SLF4J que trabaja con LOG4J y utiliza la licencia Apache 2.0.

- El compilador necesita ANTLR para revisar la sintaxis EPL. La librería `antlr-runtime` es necesaria para el runtime.
- El compilador genera código y compila el código generado usando `Janino`. Ambos funcionan bajo una licencia BSD.

La interacción con Esper se realiza compilando e implementando módulos que contienen declaraciones, enviando eventos, avanzando el tiempo y recibiendo la salida por medio de *callbacks* o sondeando los resultados actuales.

El runtime en Esper funciona como un contenedor de *statements* o declaraciones (Figura 2.2).

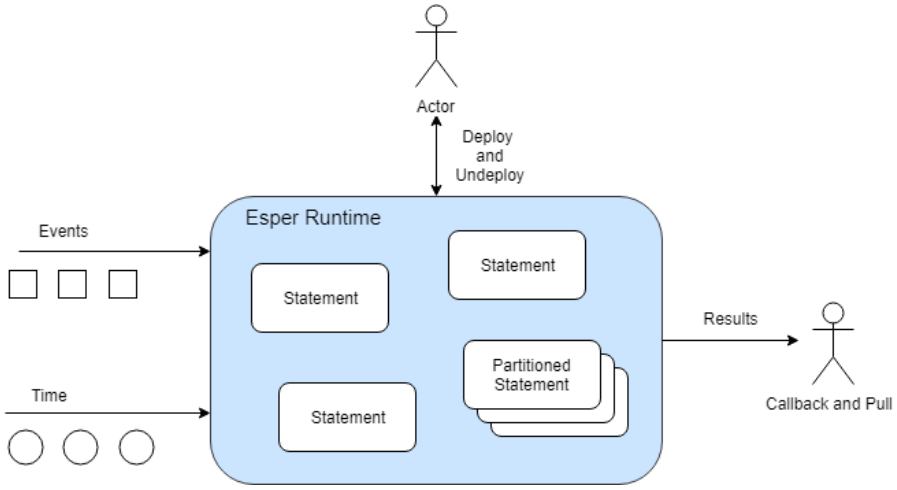


Figura 2.2: Runtime de Esper como contenedor de statements [25].

Esper soporta eventos de tipo Java Object (POJO/Bean u otro), `java.util.Map`, Object-array, JSON, Apache Avro y XML DOM. La interfaz envía eventos del tipo escogido al runtime mediante métodos `send` y devuelve eventos en el formato especificado. Para ello se deben crear *schemas* con una cláusula específica para cada caso.

La representación de eventos tiene ventajas y desventajas dentro del motor. La tabla 2.1 presenta las características de los distintos formatos.

	Java Object (POJO/Bean u otro)	Map	Object-array	JSON	Avro	Documento XML
Rendimiento	Excelente	Bueno	Excelente	Excelente	Muy bueno	No comparable y depende del uso de XPath
Uso de Memoria	Poco	Medio	Poco	Poco	Poco	Depende de las implementaciones DOM y XPath usadas puede ser alto
Método de llamada en evento	Sí	Sí, si contiene Objeto(s)	Sí, si contiene Objeto(s)	No	No	No
Prop. Anidadas, Indexadas, Mapeadas y Dinámicas	Sí	Sí	Sí	Sí	Sí	Sí
Sintaxis de eventos de grano grueso	Sí	Sí	Sí	Sí	Sí	Sí
Representación Insert-into	Sí	Sí	Sí	Sí	Sí	No
Sintaxis Create-schema	Sí	Sí	Sí	Sí	Sí	Sí
Objeto es Auto-descriptivo	Sí	Sí	No	Sí	Sí	Sí
Supertipos	Múltiple	Múltiple	Simple	Simple	Simple	No

Cuadro 2.1: Comparación de Representaciones de eventos [24].

Las declaraciones se compilan e implementan en runtime, y los resultandos se publican para los *listeners* a medida que el runtime recibe los eventos o los avances de tiempo coinciden con los criterios especificados en cada declaración.

Las declaraciones siguen la sintaxis del listado 2.1 y pueden ser consultas simples o consultas más complejas. Una consulta simple contiene solo una cláusula `select` y una única definición de stream. Se pueden construir consultas complejas que puedan relacionar múltiples streams, utilizando una cláusula `where` con condiciones de búsqueda, etc [26].

```
[annotations]
[expression_declarations]
[context context_name]
[into table table_name]
[insert into insert_into_def]
select select_list
from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]
```

Listing 2.1: Sintaxis de Esper [26].

El sistema Esper y toda la documentación están disponibles en la url principal de **Espertech**: http://www.espertech.com/

2.2.2. Siddhi

En el idioma Cingalés (de Sri Lanka) Siddhi significa evento. Es una plataforma de procesamiento de streams y procesamiento de eventos complejos que se puede utilizar para crear aplicaciones completas basadas en eventos. Puede integrarse en aplicaciones Java y Python, ejecutarse como microservicios en bare-metal, VM o Docker, y ejecutarse de forma nativa a escala en Kubernetes.

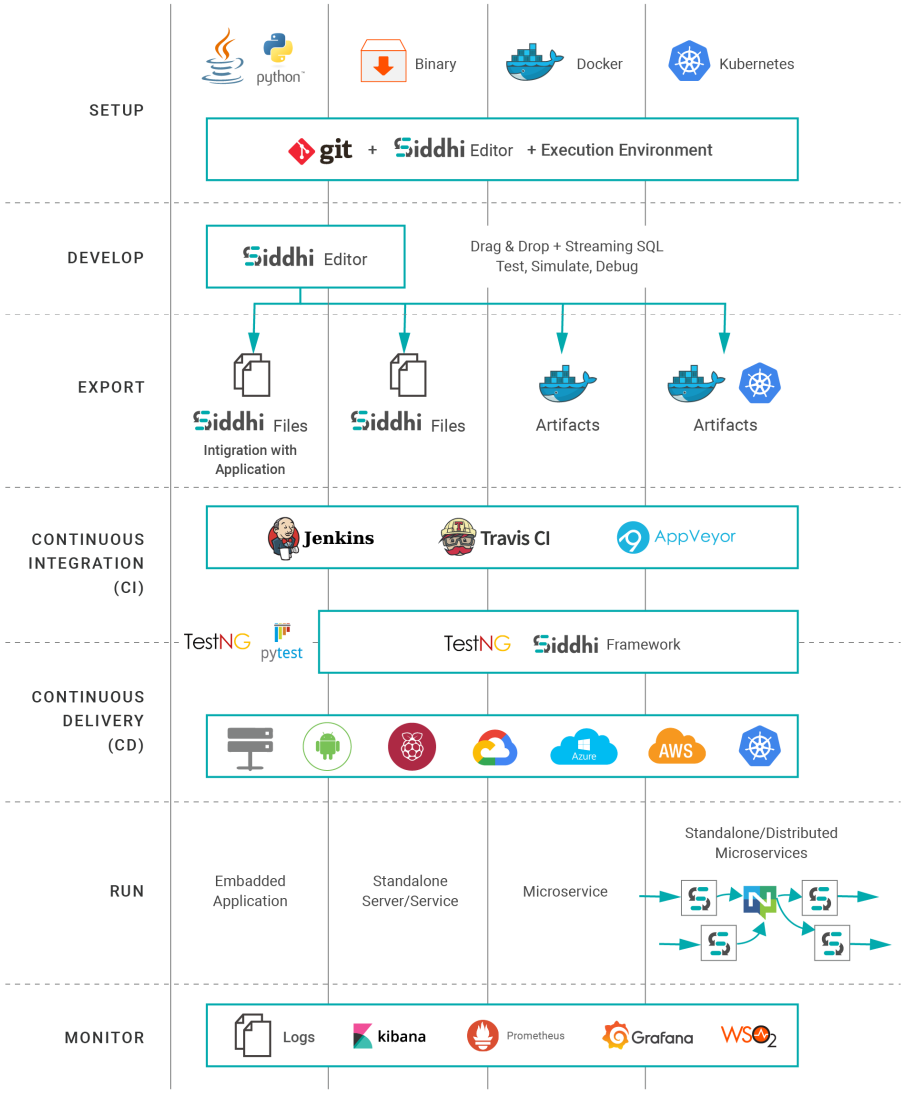


Figura 2.3: Funcionamiento de Siddhi [46].

Siddhi Application define la lógica de procesamiento de eventos en tiempo real de Siddhi como un script con una extensión de archivo `.siddhi`. Contiene sources, sinks, streams, queries, tablas, funciones y otros elementos necesarios que describen cómo se deben consumir, procesar y publicar o enviar los eventos [46].

Siddhi Streaming SQL permite escribir la lógica de procesamiento para el consumo, procesamiento, integración y envío de eventos como un script SQL.

El marco de prueba Siddhi proporciona herramientas para crear pruebas unitarias, de integración y de caja negra, para lograr un *pipeline* de *CI/CD* (integración continua y entrega continua) con un flujo de trabajo *agile DevOps*. La figura 2.4 muestra una implementación de referencia del pipeline de CI/CD de *Siddhi Apps* [7].

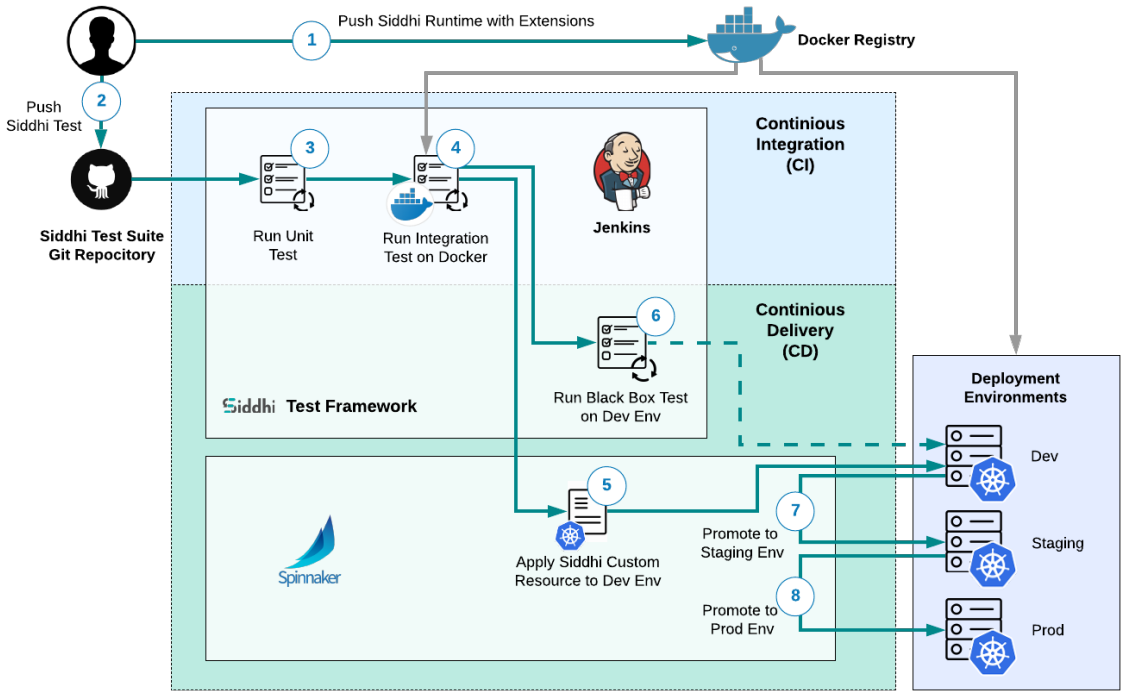


Figura 2.4: Siddhi CI/CD Pipeline [7].

Siddhi consume y envía eventos en diferentes formatos: `NATS`, `Kafka`, `RabbitMQ`, `HTTP`, `gRPC`, `TCP`, `JMS`, `IBM MQ`, `MQTT`, `Amazon SQS`, `Google Pub/Sub`, `Email`, `WebSocket`, `File`, `Change Data Capture (CDC)` (desde `MySQL`, `Oracle`, `MSSQL`, `DB2`, `Postgre`), `S3`, `Google`

`Cloud Storage`, e `in-memory`. Además soporta mensajería tipo `JSON`, `XML`, `Avro`, `Protobuf`, `Texto`, `Binario`, `clave-valor y CSV`.

Su arquitectura general es similar a la de otros motores CEP, como se muestra en la figura 2.5.

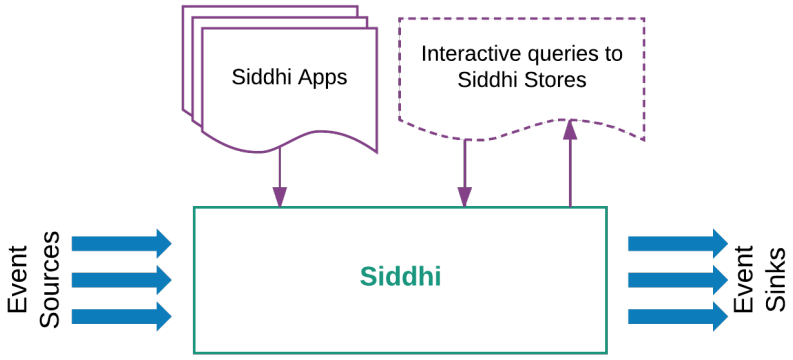


Figura 2.5: Siddhi: Arquitectura a Alto Nivel [45].

A un alto nivel, Siddhi consume eventos de varias fuentes, los procesa de acuerdo con la aplicación definida y produce resultados en los sinks suscritos. Uniendo los tipos de datos aceptados por las diferentes sources y los formatos de datos configurados para cada evento podemos tener una descripción general de la arquitectura de Siddhi (Figura 2.6).

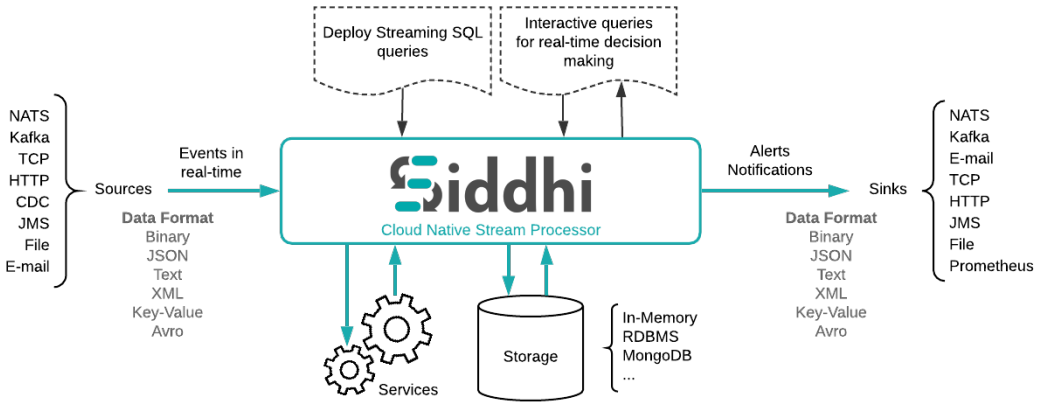


Figura 2.6: Siddhi: Descripción general [49].

La sintaxis de Siddhi se puede ver como una colección de elementos Siddhi Streaming SQL compuestos como un script, donde se definen los streams y/o las tablas, las consultas y los patrones. Esa lógica de procesamiento se almacena como una *Siddhi Application*. Aquí, cada elemento Siddhi debe estar separado por un punto y coma siguiendo la sintaxis que se muestra en el listado 2.2.

```
<siddhi app>  :
        <app annotation> *
        ( <stream definition> | <table definition> | ... ) +
        ( <query> | <partition> ) +
        ;
```

Listing 2.2: Sintaxis de Siddhi [47].

La sintaxis para definir un Stream en Siddhi se muestra en el listado 2.3.

```
define stream <stream name> (<attribute name> <attribute type>,
                             <attribute name> <attribute type>, ... );
```

Listing 2.3: Sintaxis de Siddhi Stream [47].

Una Siddhi App proporciona un entorno de ejecución aislado para procesar la lógica de ejecución. Puede implementarse y procesarse independientemente de otras Siddhi Apps en el sistema. Las Siddhi App pueden utilizar sources y sinks *inMemory* para comunicarse entre sí como se muestra en el ejemplo del listado 2.4.

```
@app:name('Temperature-Processor')

@app:description('App for processing temperature data.')

@source(type='inMemory', topic='SensorDetail')
define stream TemperatureStream (
                  sensorId string, temperature double);

@sink(type='inMemory', topic='Temperature')
define stream TemperatureOnlyStream (temperature double);

@info(name = 'Simple-selection')
from TemperatureStream
select temperature
insert into TemperatureOnlyStream;
```

Listing 2.4: Siddhi Application [48].

Finalmente cabe destacar que todas las extensiones Siddhi se publican bajo licencia *Apache 2.0*. Se puede descargar y consultar la documentación en la url principal de Siddhi: https://siddhi.io/

2.2.3. Viatra-CEP

Respaldado por *Eclipse Fundation* y como parte del *Viatra project*, se encuentran cinco componentes diferentes que abarcan varios aspectos del procesamiento de modelos. Entre ellos se encuentra *Viatra CEP*, un motor de procesamiento de eventos impulsado por modelos que emplea un modelo de runtime llamado Eclipse Modeling Framework *(EMF)* para procesar eventos. El lenguaje de procesamiento de eventos de VIATRA (VEPL) se basa en álgebra de eventos y técnicas formales como la lógica temporal y la teoría de autómatas. Gracias a su editor y a su implementación del *DSL* basada en *Xtext*, no se espera que el usuario deba diseñar los patrones de eventos [28].

Como herramienta de apoyo, Xtext es un framework para la creación de lenguajes diseñado en Java y conectado con otros proyectos de Eclipse para desarrollo de metamodelos. Proporciona soporte para desarrollar compiladores, intérpretes, editores y herramientas de tipo IDE, a través de la integración con Eclipse [43].

El flujo usual de trabajo es el mostrado en la figura 2.7

.

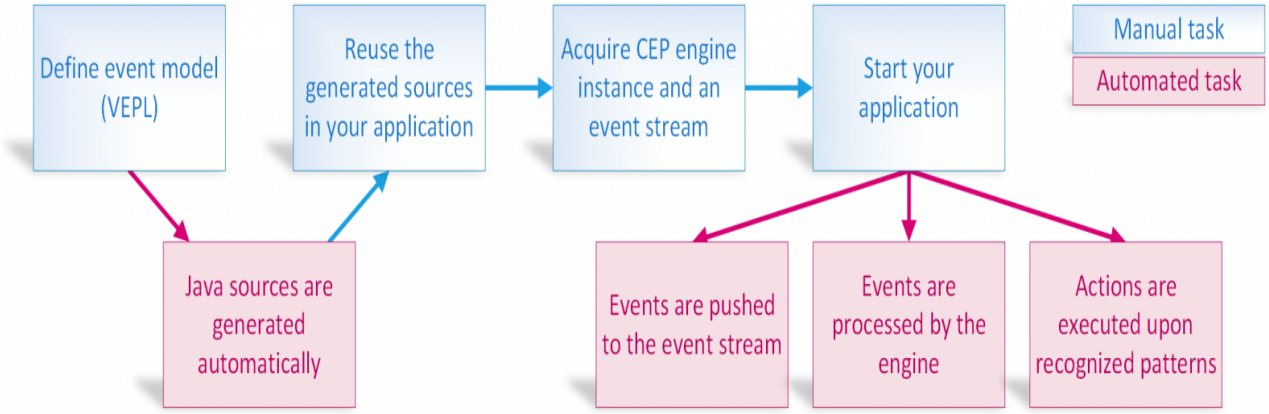


Figura 2.7: Flujo de trabajo VIATRA-CEP [28].

Primero, se define el modelo de evento diseñando los patrones de eventos simples y complejos y las reglas que definen acciones ejecutables, asociándolas con la detección de los patrones previamente definidos.

Los patrones se definen de manera gráfica a través de *EMF-IncQuery*, y de manera automatizada, se integran con el VEPL [28]. Para ello se utiliza también *The VIATRA Query Language* [19]. El listado 2.5 muestra un ejemplo de un patrón complejo.

```
pattern maximumNumberOfInstances(at : ApplicationType, max : java Integer) {
    max == max find sumNumberOfInstances(at, #);
    find sumNumberOfInstances(applicationType, max);
}

pattern sumNumberOfInstances(at : ApplicationType, n : java Integer) {
    n == count ApplicationType.instances(at, _);
}
```

Listing 2.5: Calculando el máximo número de instancias [20].

Al ser un producto asociado con Eclipse su licencia es EPL-v 2.0. Para mayor información acerca del proyecto Viatra y sus diferentes componentes se puede visitar la url: https://wiki.eclipse.org/VIATRA.

2.2.4. Apache FLink

Apache Flink es un motor de procesamiento distribuido para cálculos sobre streams de datos *limitados e ilimitados*. Cuando se trabaja con streams limitados se puede procesar consumiendo todos los datos antes de realizar cualquier cálculo. Flink se integra con todos los administradores de recursos de cluster comunes, como *Hadoop YARN, Apache Mesos y Kubernetes,* pero también se puede configurar para ejecutarse como un cluster independiente. Este motor identifica automáticamente los recursos requeridos según el paralelismo configurado de la aplicación y los solicita al administrador de recursos; en caso de fallo, Flink reemplaza el contenedor que falló solicitando nuevos recursos. Toda la comunicación para enviar o controlar una aplicación ocurre a través de llamadas REST. Esto facilita la integración de Flink en muchos entornos [51].

Para generar un programa en Flink se requieren 5 pasos básicos:

1. Obtener un entorno de ejecución.

2. Cargar y/o crear los datos iniciales.

3. Especificar transformaciones sobre los datos.

4. Especificar dónde colocar los resultados de sus cálculos.

5. Activar la ejecución del programa.

Para obtener el entorno en Java basta con invocar los métodos:

- `getExecutionEnvironment()`
- `createLocalEnvironment()`
- `createRemoteEnvironment(String host, int port, String... jarFiles)`

A continuación se leen los datos de las diferentes fuentes, y de ser necesario, se transforman los streams (listado 2.6).

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.
    getExecutionEnvironment();
DataStream<String> text = env.readTextFile("file:///path/to/file");

DataStream<String> input = ...;
DataStream<Integer> parsed = input.map(new MapFunction<String, Integer>() {
    @Override
    public Integer map(String value) {
        return Integer.parseInt(value);
    }
});
```

Listing 2.6: Lectura y conversión de streams [16].

Posteriormente, se envían los datos procesados a un sink, que consume *datastreams* y los reenvía a archivos, sockets, sistemas externos o los imprime. Finalmente se ejecuta la aplicación (listado 2.7).

```
final JobClient jobClient = env.executeAsync();
final JobExecutionResult jobExecutionResult = jobClient.getJobExecutionResult(
    userClassloader).get();
```

Listing 2.7: Ejecución de la Aplicación [16].

FlinkCEP es la librería CEP implementada sobre Flink, que cuenta con una API de patrones. Sobre dicha API se aplican los patrones individuales, combinados, grupos de patrones o de detección.

Como las aplicaciones Flink se basan en Java y Scala, la API de streams soporta diferentes tipos de datos. El serializador nativo de Flink puede operar de manera eficiente en tuplas Java y POJO así como en tuplas Scala.

Dependiendo del tipo de integración que se requiera dentro del proyecto Flink se dispone de determinados recursos, la figura 2.8 ilustra a un alto nivel de abstracción la arquitectura del proyecto.

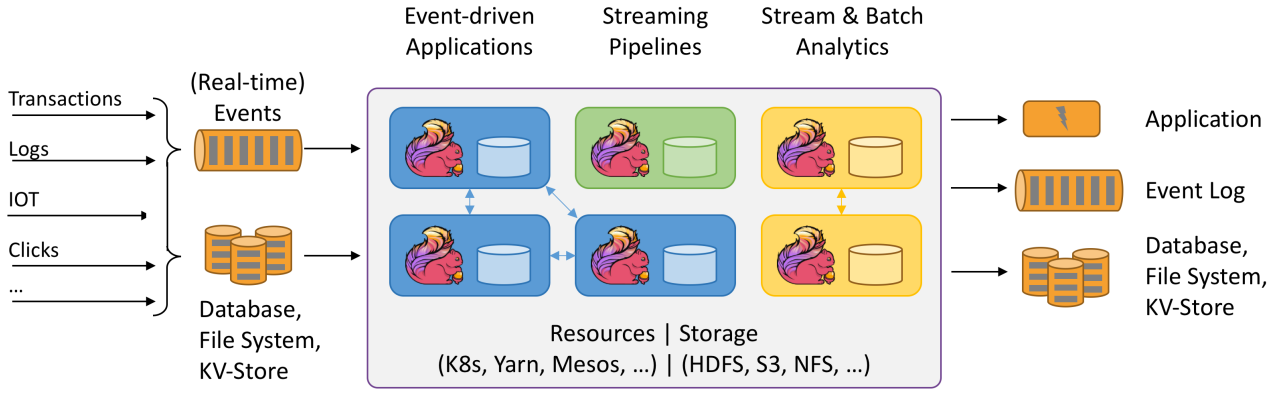


Figura 2.8: FLINK [17].

Al ser un producto de Apache su licencia actual es *Apache License 2.0*. Para obtener mayor información acerca de FlinkCEP se puede visitar su url: https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/libs/cep.html

2.2.5. BeepBeep 3

Todo en *BeepBeep* está basado en el concepto de *pipeline* como se muestra en la figura 2.9. El procesamiento de streams se realiza instanciando cajas sencillas llamadas *processors*, similares al concepto matemático de transductores [23] y conectando las salidas de algunos processors a las entradas de otros para crear una cadena de procesamiento.

Un processor es un objeto que toma cero o más streams de eventos, y produce cero o más streams de eventos como salida. Existen 2 formas de interactuar con un processor:

- *Pullable y pushable* son dos interfaces para recibir y dar eventos desde/hacia un pro-

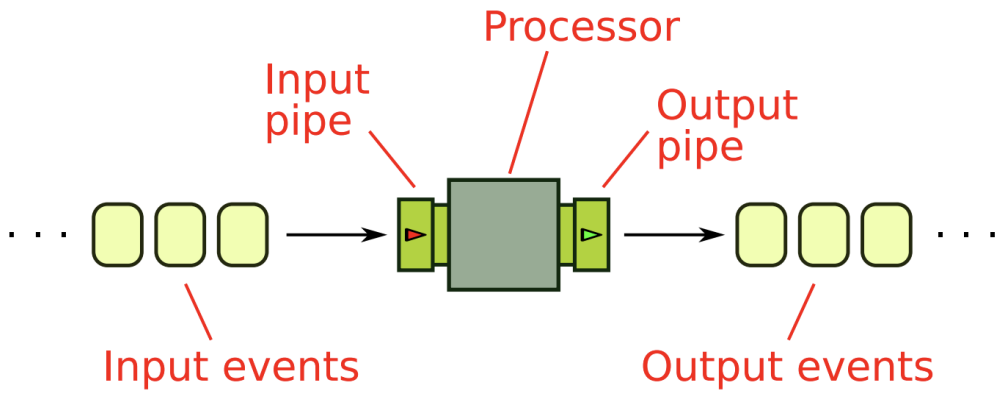


Figura 2.9: Beep Beep:Representación Gráfica [22].

cessor, respectivamente.

- El *conector* proporciona métodos estáticos para vincular fácilmente la salida de un processor a la entrada de otro.

Las instrucciones de configuración utilizan el entorno de desarrollo integrado (IDE) de Eclipse, pero se pueden transferir fácilmente a otros IDE, o incluso a una instalación mediante línea de comandos.

Utiliza Java Development Kit (JDK) para compilar. BeepBeep está desarrollado para cumplir con la versión 6 de Java.

BeepBeep divide el procesamiento de un seguimiento de eventos en dos partes.

- *Trace Manipulation Functions* (TMF) son unidades de procesamiento que manipulan eventos sin acceder a su contenido. Un ejemplo simple de TMF sería un processor que devuelve cada enésimo evento de una traza de entrada y descarta los demás. En tal caso, el contenido real de cada evento es irrelevante.

- *Event Manipulation Functions* (EMF) son unidades de procesamiento que leen o escriben datos hacia/desde un evento individual. A diferencia de las TMF, las EMF son específicas del tipo de evento que se está manipulando. BeepBeep incluye EMF para

manipular conjuntos, documentos XML y tuplas; y permite a los usuarios crear sus propias funciones para cualquier objeto de evento especial que deba procesarse.

Inicialmente, por su modularidad maneja objetos Java y al unir processors se programa. Sin embargo, otra forma de crear consultas es usando el lenguaje de consulta de BeepBeep, llamado *Event Stream Query Language* (eSQL); eSQL es un lenguaje de dominio específico codificado por software, con la posibilidad de ser completamente rediseñado por el usuario, para poder instanciar una cadena de processors desde un árbol de análisis.

BeepBeep es un software gratuito y de código abierto, distribuido bajo la Lesser General Public License (LGPL). Para obtener mayor información acerca de Beep Beep 3 se puede visitar su url: https://liflab.github.io/beepbeep-3/index.html

Además de los mencionados anteriormente, algunos motores CEP que pueden servir como referencia son *ODE [18], SASE [13]* y *Cayuga [6]*, entre otros.

CAPÍTULO 3

ESTADO DEL ARTE

Teniendo en cuenta que la motivación de este desarrollo es lograr comparaciones del rendimiento de distintos motores CEP, para que el usuario pueda tomar la mejor decisión a la hora de escoger el que mejor se acomode a sus necesidades, en este capítulo se revisan los principales trabajos relacionados con comparación de rendimiento de estos motores y hasta donde se ha avanzado en este campo.

Wu, Diao y Rizvi [52] realizaron una comparación entre *SASE* y un procesador de flujo relacional llamado *TelegraphCQ*, desarrollado en Berkeley. En este estudio, SASE se comporta mucho mejor que TelegraphCQ, pues usa NFA para capturar naturalmente los eventos de secuenciación y el algoritmo *Partitioned Active Instance Stack* (PAIS) para manejar la prueba de equivalencia durante la ejecución de NFA. En cuanto a escalabilidad muestra unos resultados mucho mejores. Sin embargo, SASE no puede manejar jerarquías de tipos de eventos complejos; es decir, que la salida de una consulta no se puede usar como entrada de otra. Por este motivo no es conveniente su uso actual en el mundo de los eventos complejos.

La Universidad Cornell desarrolló *Cayuga*, un sistema CEP de propósito general que se puede utilizar para detectar patrones de eventos en múltiples streams no relacionados, lo cual soluciona el problema presentado con SASE [6].

Este motor de procesamiento, de un solo subproceso, lee los streams y los procesa mediante autómatas con buffers lo que ayuda con el almacenamiento de datos de entrada, lo cual permite comparar nuevas entradas con eventos encontrados previamente. No todas las consultas de Cayuga pueden ser implementadas por un solo autómata, y para procesar consultas arbitrarias, Cayuga admite la *re-suscripción*. Esta arquitectura basada en el modelo de re-suscripción requiere que cada resultado de la consulta se produzca en tiempo real para que el siguiente proceso lo utilice; por lo tanto, cuando se procesa una secuencia de consultas para el mismo evento, puede haber otras consultas esperando que podrían producir un resultado final más útil en poco tiempo.

Para manejar este problema, Cayuga usa colas de prioridad; sin embargo, dado que el núcleo es de un solo subproceso, las mejoras de rendimiento obtenidas a través de estas optimizaciones no fueron importantes.

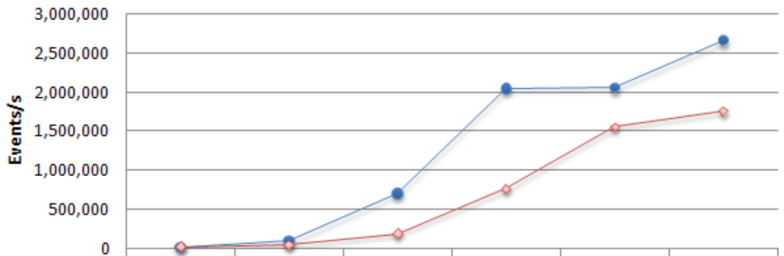
Al igual que Cayuga, Esper también tiene la capacidad de expresar condiciones complejas de coincidencia que incluyen ventanas temporales, unión de diferentes flujos de eventos, así como filtros y ordenación. Se realizó una comparación exhaustiva a nivel teórico sobre sistemas de procesamiento de datos y eventos como *Esper, Oracle CEP, TelegraphCQ, NiagaraCQ o CQL/Stream* [9], teniendo en cuenta sus modelos funcionales y de procesamiento, sus modelos de implementación e interacción y sus modelos de datos, tiempo y reglas (ver Tablas I y III de [9]). Los autores muestran una comparación en todos los aspectos mencionados anteriormente; y a pesar de no evidenciar resultados comparativos acerca del rendimiento, sus comparaciones teóricas ayudan a determinar los sistemas de procesamiento de eventos en los que centrar la atención.

También se han realizado pruebas comparativas de rendimiento de Esper y Siddhi [50]. En este trabajo seleccionaron una máquina servidor con procesador Intel(R) Xeon(R) X3440 de 2.53GHz, 4 núcles, 8M de caché, 8GB de RAM y sistema operativo Debian sobre un Kernel Linux 2.6.32-5-amd64. En cada caso se enviaron de 10 a 1000.000 eventos a través del sistema, midiendo el rendimiento del procesamiento.

Ambos motores CEP se comportan de manera estable a lo largo de la carga de trabajo, todos los mensajes se envían a los motores CEP utilizando un bucle `for`, sin demora, para que se procesen y envíen los eventos. En dos casos sencillos de filtrado y comparación (Figura 3.1 y Figura 3.2), Siddhi lo hace un 20-30 % mejor; en un tercer caso (Figura 3.3), Siddhi se muestra de 10 a 15 veces más rápido. Se concluye que ambos motores CEP presentan muy buen rendimiento al procesar alrededor de 0,3 millones de mensajes por segundo para casos complejos, y hasta 2 millones de mensajes por segundo para casos simples. Sin embargo, Siddhi se comportó mejor en todos los casos.

```
1 select symbol, price from StockTick(price>6)
```

Listing 3.1: Simple filter EPL query [50].



# of Events	10	100	1,000	10,000	100,000	1,000,000
Siddhi	21,739	92,593	704,225	2,053,388	2,061,856	2,662,336
Esper	16,667	36,900	185,185	760,456	1,560,062	1,752,265

Figura 3.1: Siddhi Vs Esper Simple Filter Query Comparison [50].

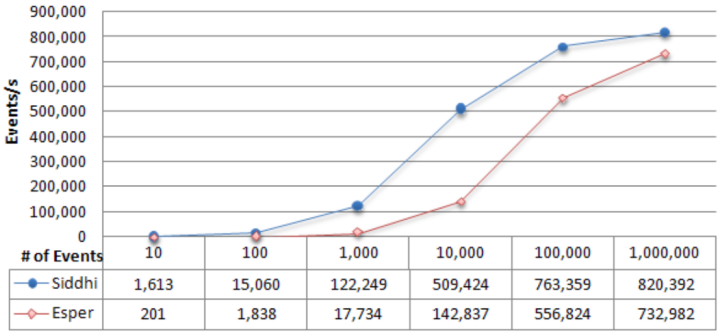
```
1 select irstream symbol, price, avg(price) from StockTick(symbol='IBM').win
    :time(.005)
```

Listing 3.2: Average over time window EPL query [50].

```
1 select f.symbol, p.accountNumber, f.accountNumber
2 from pattern [every f=FraudWarningEvent2 -> p=PINChangeEvent2(
    accountNumber = f.accountNumber)]
```

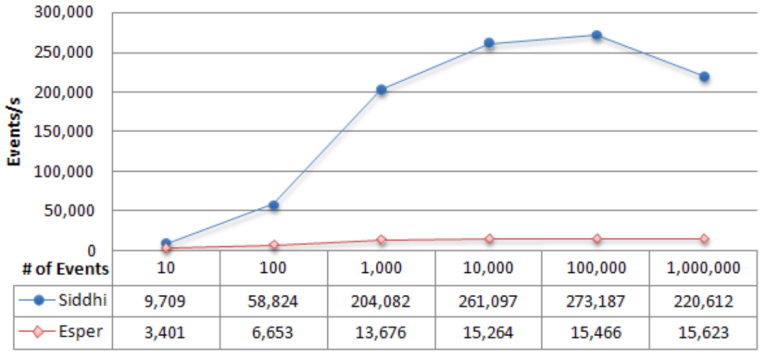
Listing 3.3: State machine pattern matching EPL query [50].

A pesar de los hallazgos, esta comparación se realizó en una máquina específica en un momento determinado. El cambio en las versiones tanto de Siddhi, como de Esper, así como los demás motores CEP, pueden variar los resultados. Además, el formato de datos,



# of Events	10	100	1,000	10,000	100,000	1,000,000
Siddhi	1,613	15,060	122,249	509,424	763,359	820,392
Esper	201	1,838	17,734	142,837	556,824	732,982

Figura 3.2: Siddhi Vs Esper Average over Time Window Query Comparison [50].



# of Events	10	100	1,000	10,000	100,000	1,000,000
Siddhi	9,709	58,824	204,082	261,097	273,187	220,612
Esper	3,401	6,653	13,676	15,264	15,466	15,623

Figura 3.3: Siddhi Vs Esper State Machine Query Comparison. [50].

el formato de eventos y los patrones requeridos pueden alterar el comportamiento de los motores; por lo cual, es necesario un software que mida estas variaciones y pueda realizar comparaciones en tiempo real.

Existen algunos benchmarks que analizan el rendimiento de aplicaciones que utilizan procesamiento de datos [40], sin embargo, dichos estudios individuales no presentan alguna aplicación disponible que compare el rendimiento de los motores CEP.

En el estudio [35] se utiliza Esper para medir el rendimiento de los sistemas ejecutando operaciones comunes. Se monitoriza el sistema a nivel de microarquitectura utilizando el generador de perfiles Intel VTune® y se recopilan métricas a nivel de aplicación, como el consumo de memoria y el rendimiento máximo sostenido, demostrando cómo las estructuras de datos alternativas pueden mejorar drásticamente el rendimiento y el consumo de recursos

en los sistemas CEP, buscando una mejora en el desempeño en los procesadores.

A raiz de estos desarrollos y como parte del proyecto *BiCEP* [11] nace *FINCoS* [36], en su última versión (2.4.2) disponible. FINCoS se puede utilizar no solo para estudios de rendimiento independientes, sino también como componente de medición y generación de carga reutilizable en kits de pruebas de rendimiento.

Este marco consta de 5 componentes principales (ver figura 3.4):

- Los *drivers* que simulan fuentes de eventos externas
- Los *sinks* que reciben los eventos generados por el sistema bajo prueba
- Los *adapters* que convierten los eventos entregados por la aplicación al formato de datos requerido por el sistema bajo prueba o viceversa
- Una aplicación gráfica llamada *controller* la cual permite al usuario configurar, ejecutar y monitorizar el resultado de las pruebas
- Un *performance monitor*, el cual permite visualizar los resultados tanto en tiempo real como después de la finalización de la prueba

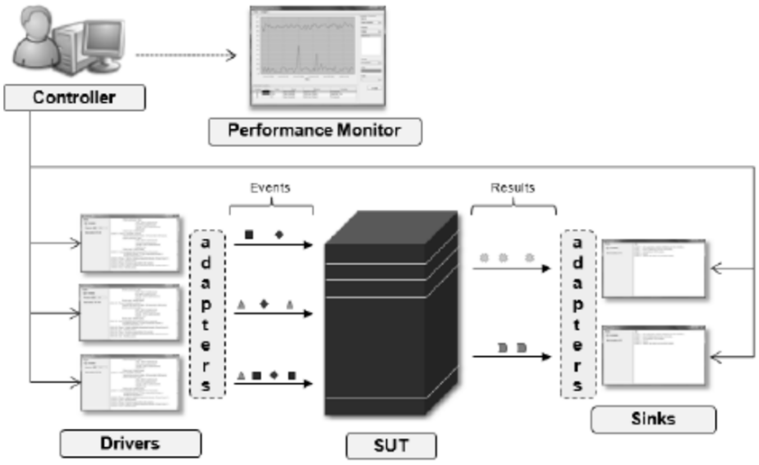


Figura 3.4: Vista general de los componentes de FINCoS [36].

La ejecución de los drivers se puede dividir en fases, cada una con sus propias características de carga de trabajo, como por ejemplo, la tasa de envío de eventos o tipos y conjuntos

de datos. El usuario puede escoger si los datos los generará la aplicación o serán cargados desde un *dataset* específico de eventos reales.

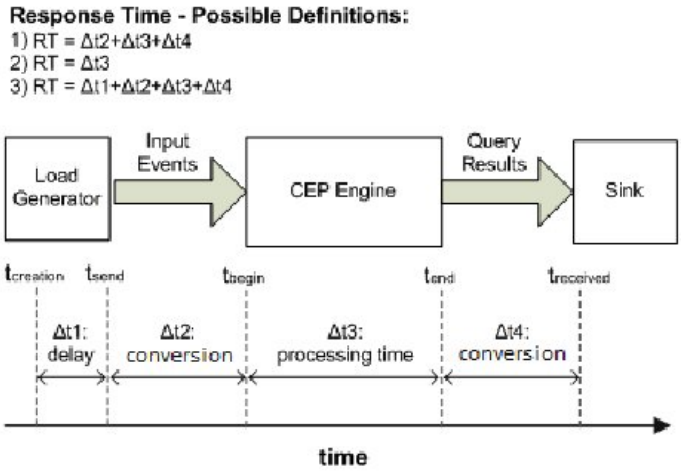


Figura 3.5: Modos de medición de latencia soportados por FINCoS [36].

Mediante los adaptadores, el sistema permite realizar pruebas sobre diferentes motores CEP, utilizando la API de cada producto; la versión 2.4.2 de FINCoS realiza conexión con Esper (versión 4.9).

Una vez finalizada la prueba, el rendimiento del sistema sometido a pruebas se mide mediante el performance monitor y los archivos de registro (.log) producidos por los sinks.

FINCoS también permite medir el rendimiento mientras las pruebas se ejecutan a costa de una sobrecarga. La herramienta presenta estadísticas de rendimiento en formatos tabulares y gráficos; el primer formato muestra un *snapshot* del rendimiento y la latencia para cada consulta que se ejecuta en el sistema bajo pruebas, mientras que el segundo muestra la evolución de estas métricas a lo largo del tiempo.

También permite configurar los tiempos de evaluación en nanosegundos o milisegundos, y su latencia puede ser tomada dependiendo del punto de medición capturado (ver figura 3.5). Según el punto de medición capturado su modo puede ser *end-to-end*, el cual representa el tiempo que tarda una tupla de salida en llegar a un sink después de que un driver envía el evento que lo desencadenó. Alternativamente, el segundo modo se puede utilizar si el usuario

desea medir solo el tiempo de procesamiento de eventos dentro del motor CEP; en este caso, los eventos tienen una marca de tiempo dentro de los adaptadores, inmediatamente antes y después de enviar y recibir eventos al motor CEP. La última versión encontrada requiere configurar algunos archivos manualmente para recibir los streams y las queries, y la versión que utiliza de Esper está desactualizada. FINCoS es un punto de referencia muy apropiado para continuar con la comparación de motores CEP.

Automatizando las interfaces, agregando más adaptadores para la comunicación con otros motores, uniendo el performance monitor con la GUI principal, y actualizando la versión tanto de Esper como de FINCoS se conseguirá una aplicación actualizada y adaptativa que genere estadísticas y pruebas de rendimiento con diferentes motores CEP.

CAPÍTULO 4

FINCOS++

El objetivo principal de este trabajo consiste en conseguir una herramienta que analice diferentes motores CEP de manera flexible y neutral, y que evalúe sus tiempos de procesamiento, no solo para que el usuario pueda elegir un motor de procesamiento de eventos que se adapte mejor a sus necesidades, sino para que los desarrolladores puedan comparar y mejorar el rendimiento de los mismos.

Teniendo en cuenta que los motores CEP evaluados son de código abierto, hemos considerado que una solución efectiva consiste en tomar como punto inicial la última versión de FINCoS referenciada en el capítulo 3. A partir de ella, se ha desarrollado una nueva versión, *Fincos++*. En este capítulo se describe la arquitectura de FINCoS++, su diagrama de clases, las técnicas de programación utilizadas para optimizar su funcionamiento, y las mejoras incluidas en esta nueva versión, así como los pasos ejecutados para llegar a dichas mejoras.

4.1. Arquitectura del sistema

El sistema FINCoS++ mantiene la misma estructura que la última versión de FINCoS. Sigue contando con 5 componentes principales y un proceso separado (daemon service) que se ejecuta en segundo plano y se comunica con la aplicación principal mediante RMI, con

algunas mejoras en su funcionalidad y con un grado mayor de automatización.

En los siguientes apartados se detallan los cambios realizados sobre estos componentes.

4.1.1. Sources

Los sources envían los eventos previamente configurados por el usuario al motor CEP bajo prueba, ya sea sintéticamemte (simulando una fuente externa) o desde archivos de datos previamente cargados de alguna fuente externa real.

En la versión original de FINCoS, los llamados drivers permitían la configuración de diferentes fases de eventos de manera sintética; sin embargo, los esquemas de eventos los debía configurar manualmente el usuario accediendo a un archivo de configuración `XML`. Este tratamiento manual permitía introducir errores muy fácilmente. En FINCoS++ se han incluido nuevas clases para automatizar dicho proceso sin necesidad de la intervención del usuario dentro del código fuente, para que los streams de eventos sean enviados a los motores CEP conectados.

4.1.2. Sinks

Los sinks reciben los eventos de salida resultantes de la ejecución de patrones y consultas efectuadas en los motores CEP. Los resultados se almacenan en archivos `.log` para la posterior validación y medición.

Al igual que los esquemas, en FINCoS los patrones también debían ser escritos manualmente en un archivo `XML` dentro de una carpeta de configuración. FINCoS++ incluye un menú, donde se configura el patrón correspondiente, se escoge el motor CEP al cual va destinado, y se configura el esquema de eventos de salida requerido por FINCoS++.

4.1.3. Controller GUI

El componente *Controller GUI* contiene la interfaz de usuario FINCoS++, donde se pueden configurar todos los aspectos del entorno de ejecución; el número de sinks, de sources, los motores CEP a los cuales están destinadas las pruebas, los diferentes esquemas y patrones

de eventos; las funciones de carga, ejecución, parada, interrupción de componentes y de datos; y el acceso al performance monitor tanto en ejecución como durante la medición offline. En FINCoS++ se ha añadido en la interfaz de usuario un acceso al performance monitor offline, que no estaba incluido en la última versión de FINCoS.

4.1.4. Adapters

Los adapters convierten los eventos producidos por FINCoS++ al tipo de evento soportado por el motor CEP a evaluar y viceversa. Básicamente permiten la conexión entre las sources y los sinks con los motores CEP.

4.1.5. Performance Monitor

El performance monitor es el encargado de recopilar las métricas realizadas por FINCoS++ para su visualización y análisis por parte del usuario. Se puede realizar en tiempo real durante la ejecución de la prueba, o puede mostrar las estadísticas después de la ejecución (modo offline) utilizando los archivos `.log` producidos por los sinks.

Este componente tiene dos modos de visualización: Gráfica mostrando las estadísticas sobre una línea de tiempo, o en formato de tabla mostrando los resultados como promedios, mínimos, máximos y/o resultados finales dependiendo del dato requerido.

4.1.6. Daemon Service

Este componente se utiliza en FINCoS para la comunicación entre la aplicación principal y el motor CEP que se está evaluando, que puede estar en otra máquina. Para ello, utiliza las librerías del motor CEP para comunicarse con éste, pero se ejecuta como proceso independiente de la aplicación FINCoS.

Aunque debe ejecutarse como una aplicación independiente, el *daemon service* encapsula sources y sinks. Esta aplicación se ejecuta en segundo plano en todas las máquinas donde se espera que se ejecuten sources y/o sinks, escuchando las llamadas *RMI (Remote Method Invocation)* desde el componente controller GUI.

Con FINCoS se logran configurar varias sources y sinks simultáneamente y en diferentes máquinas. Esto permite ejecutar las pruebas, aumentando la carga sobre el sistema bajo evaluación.

En FINCoS la comunicación se realiza sobre un único motor CEP, Esper, mientras que en FINCoS++ se ha extendido este componente de forma que, se pueden ejecutar pruebas sobre otro motor al mismo tiempo, Siddhi. Esto permite comparar estos motores en tiempo real, o en la modalidad offline, analizando detalladamente las estadísticas recogidas por los sinks en los archivos `.log`, utilizando varias o incluso la misma máquina en una única ejecución.

Como se mencionó en la Capítulo 3 se pueden tomar mediciones en dos modos.

- *End To End:* Desde que la source envía el evento generado hasta que el sink recibe la salida procesada.
- *Process Time:* Tiempo medido dentro de los adaptadores desde que el adaptador es notificado de la llegada de un evento, hasta que el evento que causa la salida es enviado al motor CEP.

4.2. Tecnologías Utilizadas

Para crear una herramienta capaz de comunicarse con varios motores CEP, que genere eventos sintéticos en tiempo real o lanzarlos desde un archivo, y evitar el problema de los retardos en la recepción y envío de streams, es necesario hacer uso de varias técnicas. A continuación se introducen los aspectos más relevantes del desarrollo y la actualización de FINCoS a FINCoS++.

Cabe mencionar que el lenguaje de programación empleado para el desarrollo es Java. La versión original de FINCoS estaba diseñada en JavaSE_1.6 [1]; para FINCoS++ se hace uso de la plataforma OpenJDK 11.0.1 [2]. Además, el desarrollo del proyecto se ha realizado sobre la plataforma Eclipse IDE for Java Developers [14] en su versión 2019-12 (4.14.0).

Al igual que en FINCoS 2.4.2, también se requieren las librerías de Esper (v 4.9.0), JMS, HornetQ (v. 2.2.14), Javassist, JfreeChart, SwingX.

Para extender FINCoS++ con el motor de Siddhi, ha sido necesario utilizar 75 librerías, incluyendo las propias del motor, las mismas que se encuentran en el repositorio referenciado en la sección 6.2.

4.2.1. Threads

El sistema FINCoS++, como el sistema FINCoS original, está programado utilizando *threads* para la concurrencia intraproceso. Este aspecto es fundamental, pues el sistema debe interactuar con motores CEP, sources y sinks, que son inherentemente concurrentes. Por ello, la estructura de código de la aplicación está fuertemente basada en esta técnica de programación.

Además de esto, uno de los objetivos principales para la extensión implementada en FINCoS++ es la comparación de varios motores CEP; para ello es necesario crear un subproceso para "escuchar" los eventos generados por cada motor y transmitirlos al sink designado. Esta labor se asigna a la clase `OutputListener.java`, la cual hereda de la clase `Thread` [30], que comunica con cada sink empleando un *lsnrID* para cada *listener* y un *sinkInstance* para cada sink.

Cada motor utilizado por FINCoS++ tiene su propia clase Listener (`-EsperListener.java` y `SiddhiListener.java`), que heredan de `Output-Listener`.

Para que el envío y recepción de datos estén sincronizados, antes de recibir los datos, al sistema bajo prueba se le envían los eventos a través de threads identificados; ya sea con su *threadId* o por un grupo de threads. La clase `Sender.java`, la cual también hereda de la clase `Thread`, es la encargada de enviar los eventos al sistema bajo prueba.

Estos *senders* son creados en cada source en una clase llamada `Driver.java`, que tiene la configuración designada por el usuario y enviará los eventos, ya sea sintéticos o cargados mediante un archivo. Cuando dicha source inicia su ejecución mediante el método `Start`,

inicializa un nuevo thread para realizar la carga, como se puede ver en los fragmentos de código mostrados en los listados 4.1 y 4.2. Aparecen resaltadas en rojo las líneas que corresponden a la definición y arranque de los threads.

```
Override
public void start() throws InvalidStateException {
    if (this.status.getStep() == Step.READY) {
        /* Initializes a new thread to run load, so RMI thread
         * returns immediately after call. */
        new Thread() {
            Override
            public void run() {
                try {
                    ...
                    startSyntheticPhase(sPhase, i + 1);
                        } else if (drConfig.getWorkload()[i] instanceof
                            ExternalFileWorkloadPhase) {
                        ... startExternalDatasetPhase(ePhase, i + 1);
                        } ...
```

Listing 4.1: Inicialización de thread

```
private void startSyntheticPhase(SyntheticWorkloadPhase syntheticPhase, int phaseNumber)
     throws IOException, InterruptedException {
    ThreadGroup senderGroup;
    ...
    senderGroup = new ThreadGroup("Senders");
    ...
    initialThreadRate = syntheticPhase.getInitialRate() / threadCount;
    finalThreadRate  = syntheticPhase.getFinalRate() / threadCount;
    // Creates dispatcher threads
    senders = new Sender[threadCount];
    ...
}

private void startExternalDatasetPhase(ExternalFileWorkloadPhase filePhase, int
    phaseNumber)
    throws IOException {
        ...
        senders = new Sender[1];
        ...
        }
```

Listing 4.2: Creación de senders

Además del ciclo que ejecutan los datos que pasan por el motor CEP, el perfomance monitor ejecuta un thread en segundo plano que actualiza periódicamente la GUI cuando este está en modo de tiempo real recibiendo información de los threads atados a los procesos descritos anteriormente; esto permite mostrar información en pantalla sin afectar la medición en el rendimiento de los motores.

También existe una clase para procesar los archivos `.log` del sink llamada `SinkLogProcessor`,

que realiza este procesamiento en segundo plano mediante un thread, ayudándose además de la clase `LogProcessor` dentro del `OfflinePerformanceValidator`, la cual hereda de `Thread`.

4.2.2. SwingWorkers

Además de los threads detallados en el apartado anterior, en FINCoS se utilizan *swing-workers* [29], que son un tipo particular de thread para la ejecución de tareas en segundo plano dentro de la interfaz de usuario. La principal ventaja de los SwingWorkers frente a los threads genéricos está en que se facilita en gran medida la comunicación entre los threads en segundo plano y el thread principal del entorno gráfico.

A diferencia de los objetos de tipo `Thread` que implementan el método `run()`, los swing-workers deben implementar el método `doInBackground()` para realizar los cálculos en segundo plano; estos serán actualizados en la interfaz de usuario cuando haya terminado o durante el procesamiento. Esta clase funciona muy bien para que sources y sinks ejecuten sus procesos mientras la interfaz de usuario sigue estando activa.

4.2.3. Invocación remota de métodos (RMI)

Como se comentó en la sección 4.1.6, el daemon service funciona como una aplicación independiente que se comunica con la aplicación principal mediante el interfaz *RMI* de Java. RMI permite realizar llamadas a métodos de objetos remotos situados en distintas (o la misma) máquinas virtuales Java, compartiendo así recursos y carga de procesamiento a través de varios sistemas; en otras palabras, permite la concurrencia entre procesos y la distribución de los procesos en distintas máquinas.

En la clase `DaemonServer` de FINCoS++, se define un HashMap con la lista de sources y otro con la lista de sinks y se preparan para recibir las llamadas RMI. Esta clase también contiene los métodos que inicializan cada source y sink de manera remota recibiendo su alias como parámetro. Esto se logra implementando la clase `RemoteDaemonServerFunctions.java` la cual hereda de `remote` [31], una interfaz del paquete RMI de Java que sirve para identificar

interfaces cuyos métodos pueden invocarse desde una máquina virtual no local.

4.2.4. Patrón de diseño Facade

El patrón de diseño *facade* se utiliza para estructurar el entorno del sistema y dividirlo en varios subsistemas, reduciendo su complejidad. Invocada en segundo plano con la ayuda de los mencionados swingworkers, la clase `ControllerFacade.java` implementa las siguientes funciones:

1. Carga la configuración de prueba desde el archivo de configuración
2. Manteniene la lista de sources y sinks configurados
3. Comunicación RMI con sources y sinks:
   - *a*) Mantiene la lista de sources y sinks activos
   - *b*) Inicializa sources y sinks
   - *c*) Inicia, pausa y detiene el envío de carga
   - *d*) Modifica las tasas de envío de eventos

A la hora de interactuar con la GUI, se invocan métodos como `facade.getSinkList()` para obtener la lista activa de sinks o `facade.stopSink(sink)` para detener la ejecución de un sink específico.

Al igual que los threads y los swingworkers, ayuda a generar una comunicación menos complicada entre procesos, permitiéndoles menos dependencia, para lograr un desempeño óptimo. En el listado 4.3 se muestra un ejemplo de uso de este patrón de diseño.

```
1     /**
2      * Updates the configuration of an existing Sink.
3      *
4      * @param oldCfg         The old configuration of the Sink
5      * @param newCfg         The new configuration of the Sink
6      */
7     public void updateSink(SinkConfig oldCfg, SinkConfig newCfg) {
8         synchronized (sinksTable) {
9             int index = facade.getSinkList().indexOf(oldCfg);
10            if (index > -1) {
11                facade.updateSink(index, newCfg);
```

```
12      ...
13      }
```

Listing 4.3: Ejemplo del uso de facade

4.2.5. Lenguajes de consulta de los CEP

En el caso de FINCoS++ se probarán dos motores CEP. El primero es Esper, contemplado en la versión anterior. El lenguaje de procesamiento de eventos EPL de Esper es compatible con SQL, lo que garantiza un aprendizaje rápido; pero también está altamente orientado al soporte de tecnologías modernas, especificamente a objetos (más que orientado a tablas), lo que permite una extensión simple.

Para integrar este lenguaje en FINCoS++, además de crear el adaptador y el listener respectivo, se utilizan dos partes escenciales referentes al lenguaje:

- Los streams: En los cuales se recogen los atributos de cada evento y se organizan en un único archivo con formato XML llamado `Stream_Set.Xml` donde además se guarda la configuración de Esper utilizada.

- Los patrones: Representados en este lenguaje a modo de consulta SQL. Estos patrones también se almacenan en un archivo XML llamado `Query_Set.Xml`.

El segundo motor a utilizar es Siddhi, el cual funciona con el lenguaje llamado Siddhi Streaming SQL, que utiliza una sintaxis similar a SQL y anotaciones para consumir eventos. La lógica de procesamiento del programa puede escribirse utilizando Streaming SQL y guardarse como un solo archivo con la extensión `.siddhi`; sin embargo, también se puede mantener la misma lógica utilizada con Esper y dividirlo en dos archivos, uno para los streams y otro para los patrones. A este conjunto se le llama `SiddhiApp`.

El diagrama que aparece en la Figura 4.1 muestra algunos de los elementos clave de Siddhi y cómo cada evento fluye a través de dichos elementos.

Una vez definidos los streams en cada lenguaje, se pueden definir los patrones. A continuación se presenta un ejemplo sencillo que muestra la diferencia entre la sintaxis de Esper

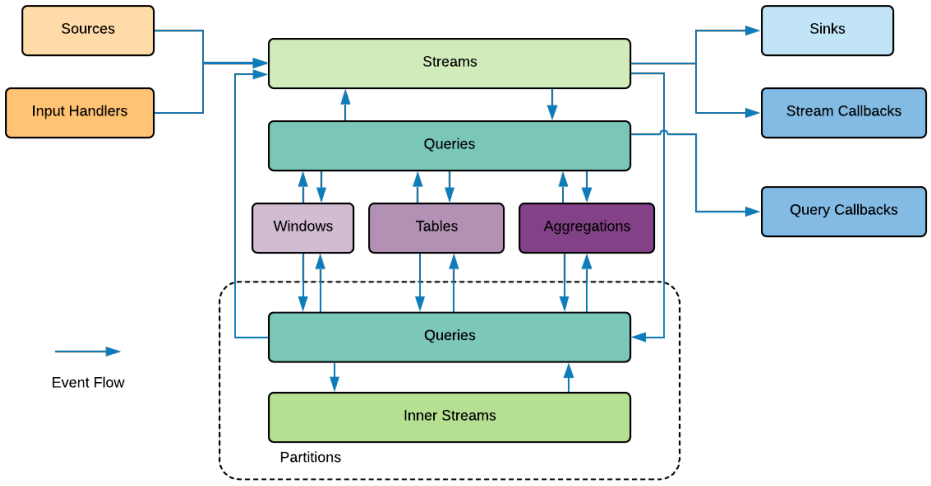


Figura 4.1: Flujo de eventos en Siddhi [49].

y la sintaxis de Siddhi:

```
insert into EjemploEsper select avg(temperatura) from SensorTemp#time(1 min);
from SensorTemp select caldera, temp * 9/5 + 32 as temp, insert into EjemploSiddhi;
```

Aunque similares, presentan diferencias en el momento de capturar sus estructuras, por lo que es necesario generar archivos de streams y de patrones diferentes para cada motor CEP.

4.2.6. Entorno gráfico

El entorno gráfico de FINCoS++ hereda de `JFrame`, tiene una interfaz principal (Figura 4.2), que cuenta con varios menús desplegables de configuración o apertura de archivos, ventanas, una barra de herramientas para controlar la ejecución de sources y sinks, así como un acceso rápido a la apertura y guardado de archivos, un panel para las diferentes sources, otro para los sinks y un panel adicional que muestra información enviada por el daemon server.

Una vez en ejecución la prueba a realizar, se despliega una ventana emergente por cada source y una por cada sink independientes de la interfaz principal. Estas ventanas muestran el estado de cada elemento (ready, running, stopped, paused, error, finished), su progreso en

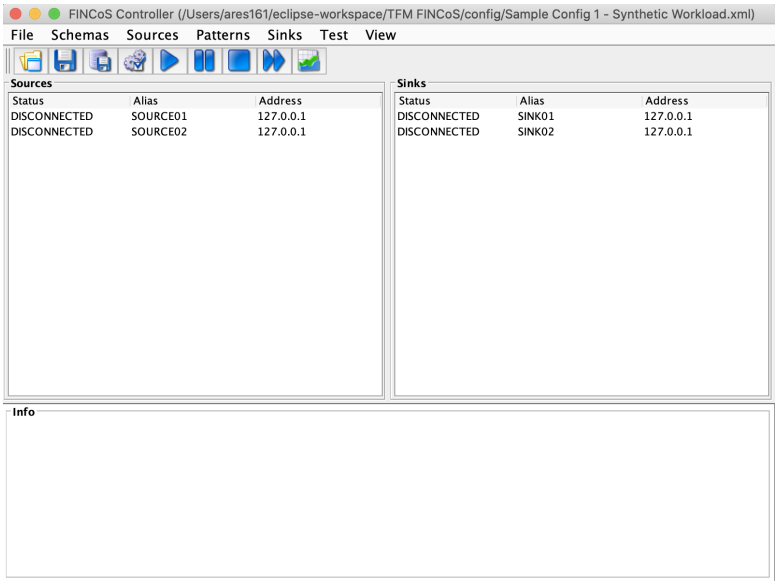


Figura 4.2: Interfaz principal de FINCoS (FINCoS Controller)

cuanto al envío y recepción de eventos, y la fase en la cual se encuentra cada source como se muestra en la Figura 4.3.

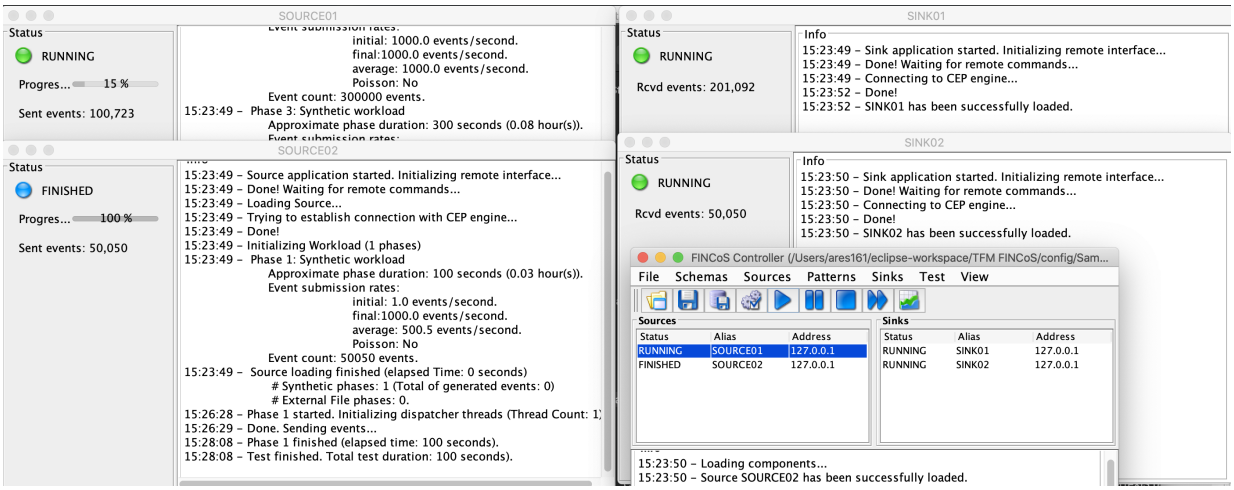


Figura 4.3: Ventanas emergentes de FINCoS

Adicionalmente cuenta con el performance monitor, el cual se puede ejecutar simultáneamente para capturar los eventos y mostrar las estadísticas en tiempo real, ya sea en forma de tabla (Figura 4.4) o de gráfica (Figura 4.5). Como se mencionó anteriormente, también se cuenta con la opción offline del performance monitor, que realiza la misma tarea a partir

de los archivos `.log` guardados después de la ejecución de una prueba.

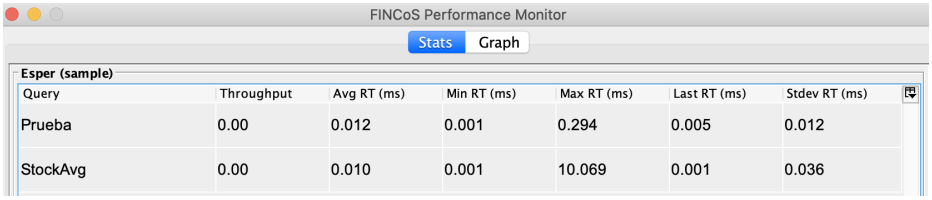


Figura 4.4: Performance Monitor (Tabla)

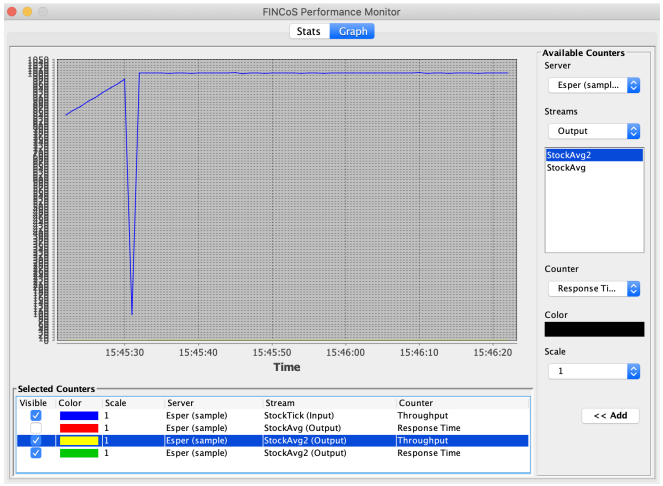


Figura 4.5: Performance Monitor (Gráfica)

4.3. Desarrollo de los objetivos

El objetivo principal del presente desarrollo es obtener un marco que compare motores CEP simultáneamente y logre generar estadísticas de rendimiento en tiempo real, así como mantenerlas en un registro para su estudio y análisis por los usuarios, para lo cual es necesario modificar varias funcionalidades y características de la versión 2.4.2 de FINCoS. Así pues, es pertinente descomponerlo en algunos subobjetivos que se detallan en los siguientes apartados.

4.3.1. Migración a Esper 8.4

Esper es uno de los motores CEP con mayor influencia durante los últimos años; sin embargo, la versión integrada con FINCoS 2.4.2 era del 2013 y estaba desactualizada, por lo que resultaba necesario migrarla a la última versión disponible. A continuación se explica de manera general el procedimiento aplicado.

Una vez añadidas las librerías necesarias para la nueva versión de Esper (antlr4, commons-compiler, janino, slf4j, log4j, javassist, entre otras) y modificado el classpath, lo siguiente es entender donde se encuentra el núcleo de configuración de la API dentro de FINCoS.

Para actualizar esta configuración es necesario editar el archivo `Stream_Set.xml`, que contiene el *Esper-configuration*; en la nueva versión es necesario definir 3 secciones: Common, Compiler y Runtime. Para esto, Esper también cuenta con una herramienta online de apoyo en la url: http://esper-config-upgrade7to8.appspot.com/eeconfig7to8/mainform.html que permite generar el nuevo esquema de configuración.

Una vez realizado este paso, es necesario actualizar los adaptadores tanto de entrada como de salida de FINCoS hacia Esper y de Esper hacia FINCoS. La clase `EsperInterface` nos permite crear la interfaz para leer los *input streams* generados por el source de FINCoS y convertirlos en streams aptos para su lectura en Esper, utilizando la configuración realizada en el paso anterior, además de enviarle los patrones y las consultas definidas en el `Query_Set.xml`.

Tras enviar los streams de eventos, es necesario actualizar el dispositivo que escuchará los eventos generados por Esper. Para esto se edita la clase `EsperListener`, que tiene los métodos de escucha y transforma los nuevos streams en streams legibles para los sinks de FINCoS. Estas tareas están descritas en la guía de actualización establecida en la documentación de Esper: http://www.espertech.com/esper/esper-information-about-esper-version-8/

Al probar el ejemplo preestablecido en FINCoS y corregir los errores con la nueva versión, se realizaron nuevas pruebas y se encontró un hallazgo interesante: La antigua versión de Esper no permitía ejecutar más de una instancia de Esper con una misma conexión, es decir,

al intentar generar ejemplos con múltiples sources y sinks conectados a Esper, solo el primero enviaba y recibía eventos a menos que se crearan diferentes conexiones al motor utilizado, incluso cuando las dos instancias en ejecución se comunicaran con Esper. En la figura 4.6 se puede observar el funcionamiento de dos instancias de Esper ejecutándose simultáneamente con la misma conexión.

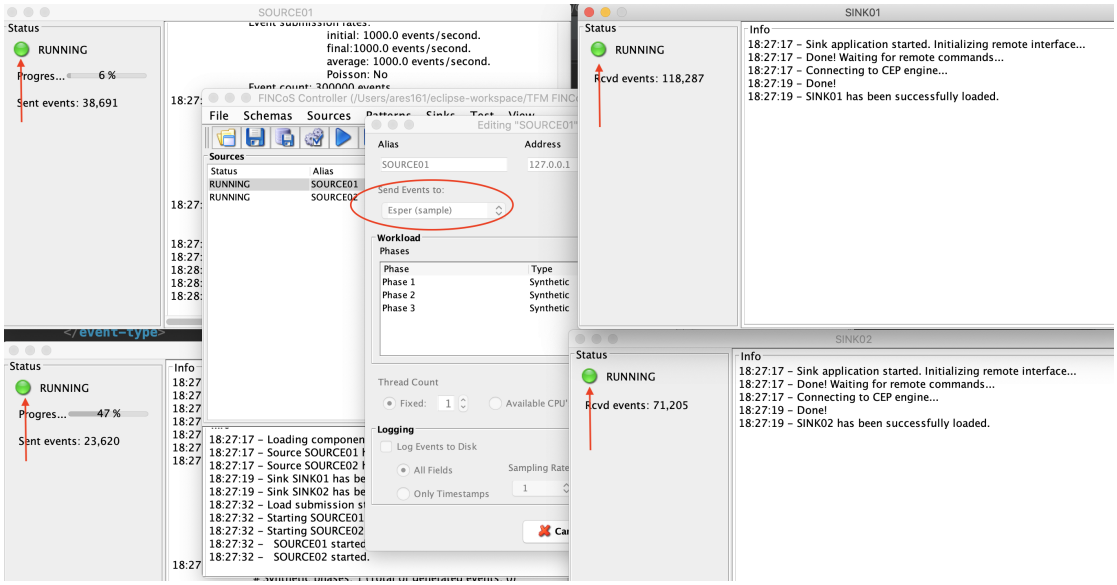


Figura 4.6: Dos Instancias de Esper corriendo exitosamente con una única conexión

4.3.2. Integración de nuevos motores (Siddhi)

Al igual que con el motor Esper, lo primero que se debe hacer para integrarlo en FINCoS++ es incluir las librerías necesarias para la ejecución de Siddhi y agregarlas al classpath dentro del entorno. Los ejemplos encontrados en la documentación de Siddhi [46] están desarrollados utilizando la herramienta de construcción de proyectos Maven. Sin embargo, como el sistema FINCoS original utilizaba la gestión de proyectos propia de Eclipse, se ha decidido continuar usando el mismo entorno de desarrollo utilizado hasta el momento y realizar adaptaciones para que funcione el motor en el entorno de Eclipse, para lo cual fue necesario adaptar más librerías al paquete de librerías de Siddhi.

A continuación, es necesario suscribir Siddhi dentro de los motores CEP soportados por

FINCoS++ en la clase `CEP_EngineFactory.java`.

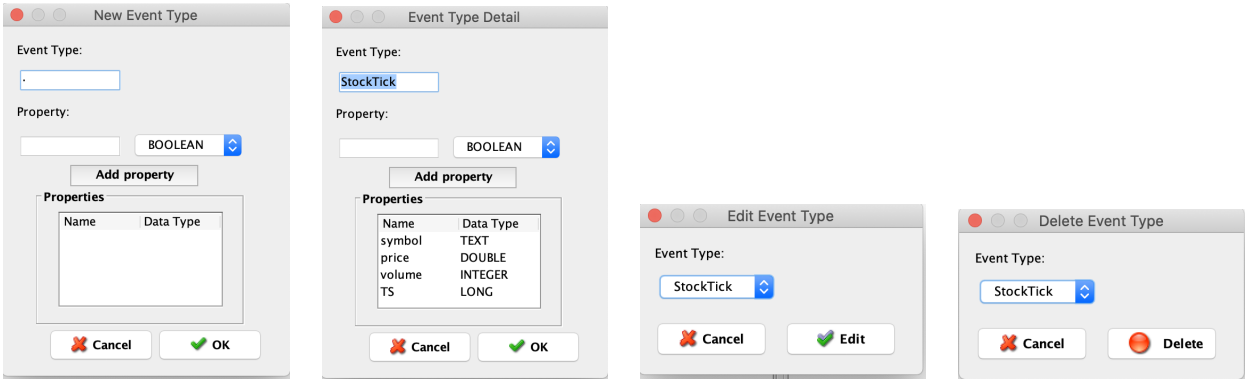
El siguiente paso es conectar los adaptadores de entrada y salida del motor a FINCoS++, para lo cual se crean las clases `SiddhiListener.java` la cual hereda de `OutputListener.java` y `SiddhiInterface.java` heredada de `CEP_EngineInterface.java`. Estas dos clases son las piezas fundamentales para la extensión de FINCoS++. Estas dos clases contienen los métodos necesarios para realizar la comunicación con el motor CEP, adaptadas para su ejecución en Siddhi, pues enviarán los streams de eventos y los patrones, y serán las encargadas de recibir los streams de Siddhi mediante `StreamCallbacks`, que serán traducidos al lenguaje de FINCoS++ utilizando el método `processIncomingEvent()`.

Los streams y los patrones se leerán de archivos de configuración XML para utilizar la misma estructura existente en FINCoS para Esper. Para esto se crean dos archivos similares a los de Esper llamados `Stream_Siddhi_Set.xml` y `Query_Siddhi_Set.xml`, que serán sobreescritos automáticamente una vez realizadas las mejoras funcionales que se describen en la sección 4.3.3.

4.3.3. Mejoras funcionales

Aún con la actualización en la versión de Esper, los streams y los patrones se continúan incluyendo manualmente en los archivos `Stream_Set.xml` y `Query_Set.xml` respectivamente, al igual que para Siddhi. Lo apropiado es realizarlo mediante la interfaz de usuario y generando una actualización automática de los archivos correspondientes. Para extender FINCoS++ con esta opción, se ha creado el menú *Schemas* (Figura 4.7). Este menú permite la creación, edición y eliminación de streams escogiendo los atributos respectivos para cada tipo de evento y sobreescribiendo el archivo `Stream_Set.xml` o `Stream_Siddhi_Set.xml` dependiendo del motor escogido.

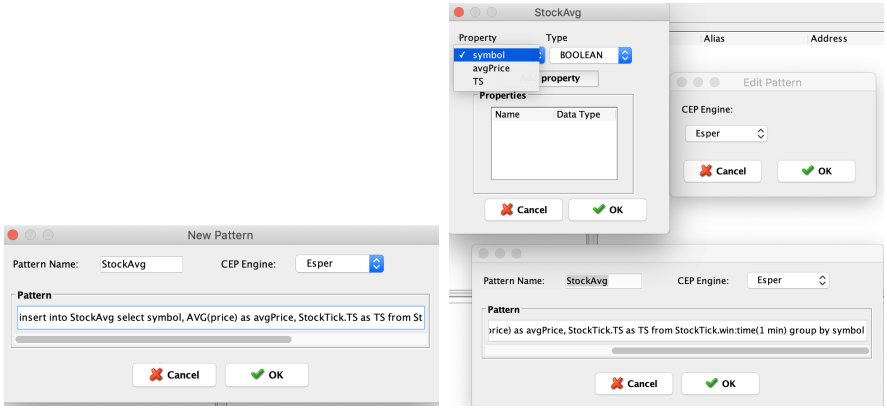
Además, para definir los patrones se ha añadido una nueva extensión a FINCoS++ mediante el nuevo menú *Patterns* (Figura 4.8), que también permite al usuario escoger el motor sobre el cual desea definir, editar o borrar el patrón y guardarlo ya sea en el `Query_-`



(a) Crear schema (b) Editar schema (c) Editar schema (d) Borrar schema

Figura 4.7: Ventanas emergentes del menú schemas

`Set.xml` o en el `Query_Siddhi_Set.xml`. Una vez definido el patrón y el motor sobre el cual se quiere aplicar dicho patrón, se despliega otra ventana emergente para asignar los tipos de datos a los atributos del stream de salida, lo cual será escrito en el *stream set* correspondiente.



(a) Crear Patrón (b) Editar Patrón

Figura 4.8: Ventanas emergentes del menú Pattern

Para que los streams y los patrones que se van creando mantengan la coherencia, fue necesario modificar algunos menús donde se introducían manualmente los esquemas que se predefinían, como es el caso del menú sink (Figura 4.9).

Otro cambio en la funcionalidad es la integración del *FINCoS Performance Monitor (Offline Mode)* (Figura 4.10) dentro de la GUI principal. Este componente permite visualizar

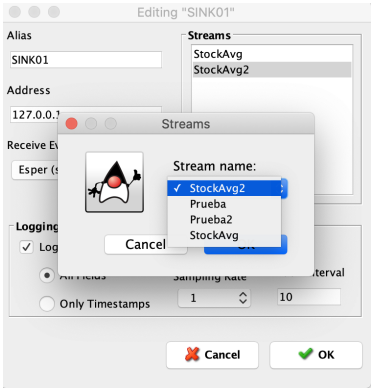


Figura 4.9: Selección de streams dentro del sink

estadísticas una vez terminada la ejecución de las pruebas para el análisis de datos. También permite guardar dichas estadísticas en el formato escogido por el usuario, pues de los archivos de registro cargados en la funcionalidad, el usuario puede escoger que datos le interesa ver en pantalla y guardar en los archivos de análisis. Esto le permite generar un informe de estadísticas en formato CSV, ideal para editar y generar aún más informes y análisis.

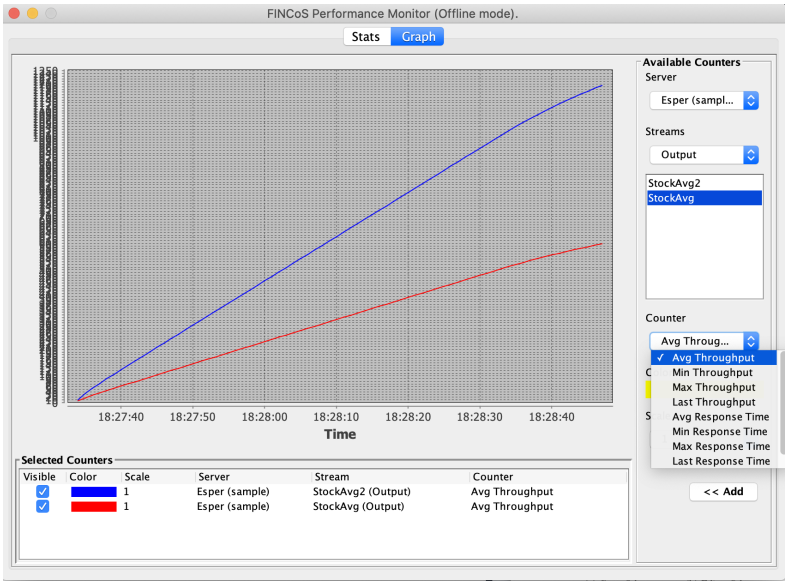


Figura 4.10: Entorno gráfico del Offline Performance Monitor

4.3.4. Gestión del proyecto

Para cumplir los objetivos en un tiempo de acuerdo al volumen de trabajo es importante no solo trazar un plan de trabajo, sino además realizar una gestión efectiva del proyecto. Para esto se recurrió a tres herramientas de gestión:

- La primera herramienta es Github, no solo como un software para control de versiones, sino como software de gestión administrativa. En este se pueden organizar tareas e hitos referentes al proyecto y pasarlos por diferentes etapas (Figura 4.11). Así se van controlando los principales aspectos del proyecto y dividiéndolos en pequeñas tareas, lo que permite seguir una línea de desarrollo. Adicionalmente, estas tareas e hitos se pueden etiquetar con un tipo específico (desarrollo, documentación, investigación, etc), además de asignarles responsable(s) y fecha límite. En este caso se utilizó Github por su versatilidad y visibilidad. Cabe destacar que cualquier software de gestión de proyectos, incluso un libro de Excel o un cuaderno, son útiles para esta tarea.

- Otra herramienta útil es una bitácora para registrar las reuniones realizadas y así mantenerse al tanto de sugerencias, modificaciones y demás avances que surgen en el proyecto. No es necesario extenderse demasiado en este punto pero se debe anotar lo más importante, lo que ayuda a mantener un registro de las actualizaciones y a recordar sugerencias y puntos importantes que sirvan como avance del proyecto y de su documentación. Como paso adicional para este proyecto en particular se incluyó dicha bitácora dentro de la Wiki del mismo repositorio.

- Finalmente, disponer de un espacio para apuntar aspectos importantes que van surgiendo a manera de anotaciones ayuda a encontrar piezas que pueden servir durante el desarrollo del mismo (hallazgos, bibliografía, dudas, pruebas fallidas y exitosas, bugs, etc). En este caso puede tratarse del detalle de las tareas anteriormente descritas y puntos o anotaciones importantes sueltas. Estas anotaciones son útiles pero pueden llevar a desviar el curso del proyecto, por lo que es importante mantener el plan de

trabajo detallado en la primera herramienta descrita.

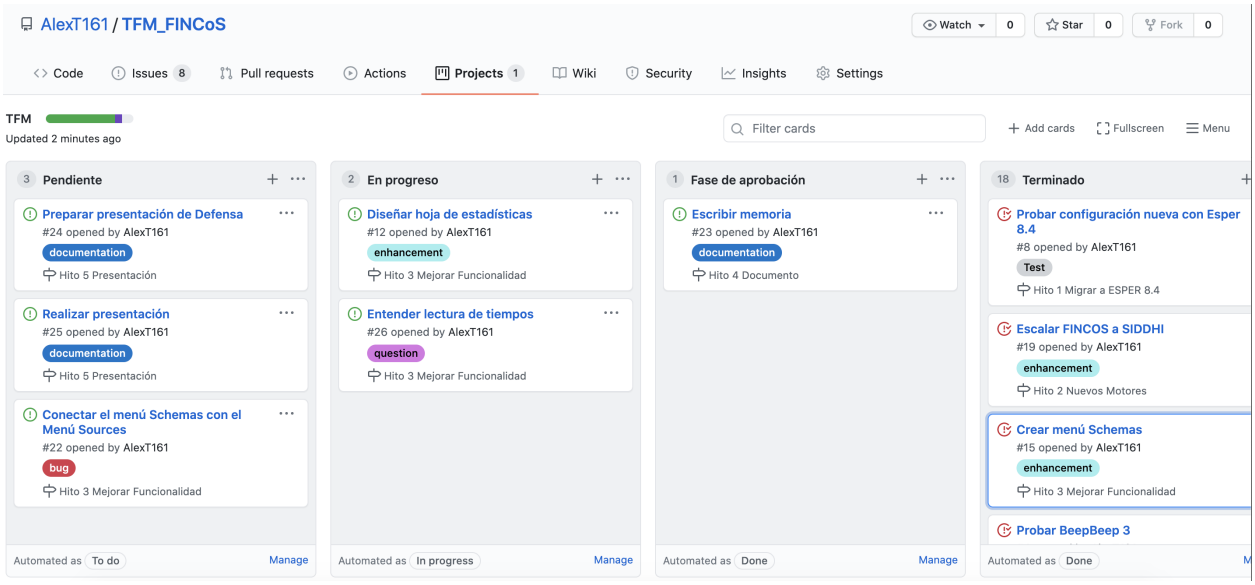


Figura 4.11: Gestión Administrativa con Github

CAPÍTULO 5

PRUEBAS Y RESULTADOS

En este capítulo se presentan las pruebas realizadas para evaluar el desarrollo de Fincos++ así como su utilidad. Para ello se ha utilizado un caso de estudio referente a la monitorización de la calidad del aire [8, 12, 34].

Las pruebas descritas en este capítulo se realizaron en un computador MacBook Pro con procesador Intel Core i5 de dos núcleos de 2.3 GHz, memoria de 8 GB a 2133 MHz LPDDR3 y sistema operativo macOS Catalina versión 10.15.4.

5.1. Configuración de experimentos

En esta sección se introducen las configuraciones utilizadas en el entorno de Fincos++ para la ejecución de los experimentos.

5.1.1. Definición de esquemas

Lo primero que se realizó fue la definición del esquema de eventos tanto para Esper como para Siddhi. El evento simple principal llamado *AirMeasurement* consta de ocho atributos: stationTs(long), stationId(integer), pm2_5(double), pm10(double), o3(double), no2(double), so2(double), co(double), los cuales fueron configurados como lo muestra la figura 5.1.

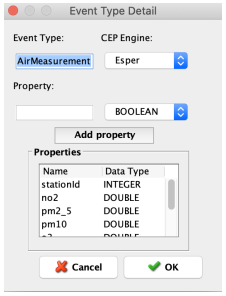


Figura 5.1: Configuración de schema AirMeasurement

5.1.2. Configuración de sources

Posteriormente, se prepararon las sources para el envío de eventos a cada motor como se puede observar en la figura 5.2; para ello, se asignó un envío de eventos durante quinientos segundos, utilizando una distribución de *Poisson* con una semilla fija de ciento cincuenta y un promedio de ochocientos a mil eventos por segundo. Este envío de eventos se realiza en una sola fase y se asignaron números aleatorios con rangos que dependen de cada atributo, como se muestra en la figura 5.3.

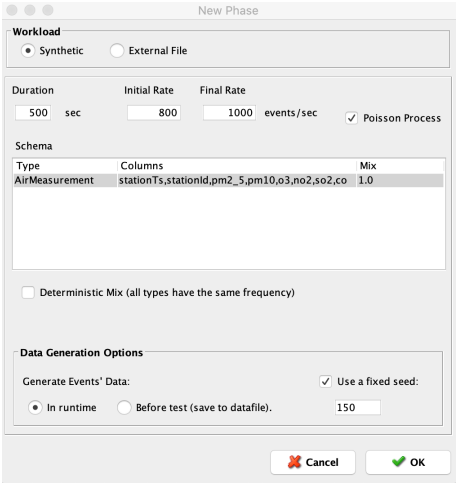
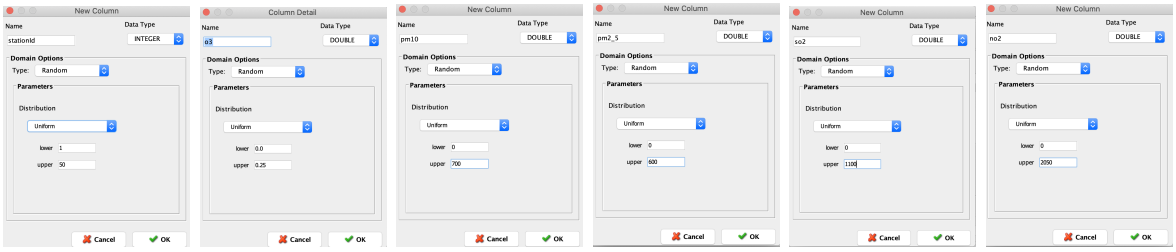


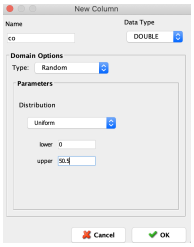
Figura 5.2: Configuración de source

5.1.3. Definición de patrones

El siguiente paso consistió en definir los patrones tanto para Esper como para Siddhi.



(a) stationId (b) o3 (c) pm10 (d) pm2_5 (e) so2 (f) no2



(g) co

Figura 5.3: Asignación de valores aleatorios sobre atributos

- Patrón para calcular el promedio de NO2 en Esper.

NO2_Avg = insert into NO2_Avg select stationTs as timestamp, stationId, avg(no2) as value from AirMeasurement.win:time(1 hour) group by stationId

- Patrones que clasifican el nivel de contaminación según la concentración en Esper.

```
<Name ="NO2_Hazardous" text = "insert into PollutantLevel select timestamp
    , stationId, 6 as levelNumber, 'NO2_Hazardous' as levelName from
    NO2_Avg (value >= 1250.0 and value <= 2049.0)" />
<Name="NO2_UnhealthyForSensitiveGroup" text="insert into PollutantLevel
    select timestamp, stationId, 3 as levelNumber, '
    NO2_UnhealthyForSensitiveGroups' as levelName from NO2_Avg (value >=
    101.0 and value < 361.0)" />
<Name="NO2_Moderate" text="insert into PollutantLevel select timestamp,
    stationId, 2 as levelNumber, 'NO2_Moderate' as levelName from NO2_Avg
    (value >= 54.0 and value < 101.0)" />
<Name="NO2_Unhealthy" text="insert into PollutantLevel select timestamp,
    stationId, 4 as levelNumber, 'NO2_Unhealthy' as levelName from NO2_Avg
     (value >= 361.0 and value < 650.0)" />
<Name="NO2_VeryUnhealthy" text="insert into PollutantLevel select
    timestamp, stationId, 5 as levelNumber, 'NO2_VeryUnhealthy' as
    levelName from NO2_Avg (value >= 650.0 and value < 1250.0)" />
<Name="NO2_Good" text="insert into PollutantLevel select timestamp,
    stationId, 1 as levelNumber, 'NO2_Good' as levelName from NO2_Avg (
    value >= 0.0 and value < 54.0)" />
<Name="AirQualityLevel" text="insert into AirQualityLevel select timestamp
    , stationId, max(levelNumber) as level from PollutantLevel.win:
```

```
   time_batch(5 minutes) group by stationId" />
```

Listing 5.1: Patrones que clasifican el nivel de contaminación según la concentración en Esper.

- Patrón para calcular el promedio de NO2 en Siddhi.

  NO2_Avg_s = from AirMeasurement#window.time(1 hour) select stationTs as timestamp, stationId, avg(no2) as value group by stationId insert into NO2_Avg_s

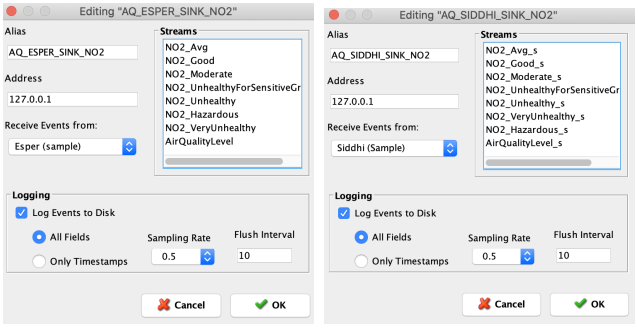
- Patrones que clasifican el nivel de contaminación según la concentración en Siddhi.

```
1 <Name="NO2_UnhealthyForSensitiveGroups" text="from NO2_Avg_s[value >= 101
     and value <361] select timestamp, stationId, 3 as levelNumber, '
     NO2_UnhealthyForSensitiveGroups' as levelName  insert into
     PollutantLevel" />
2 <Name="NO2_Moderate_s" text="from NO2_Avg_s[value >= 54 and value < 101]
     select timestamp, stationId, 2 as levelNumber, 'NO2_Moderate' as
     levelName insert into PollutantLevel" />
3 <Name="NO2_Hazardous_s" text="from NO2_Avg_s[value >= 1250.0 and value <
     2049] select timestamp, stationId, 6 as levelNumber, 'NO2_Hazardous'
     as levelName insert into PollutantLevel" />
4 <Name="NO2_VeryUnhealthy_s" text="from NO2_Avg_s[value >= 650.0 and value
     < 1250] select timestamp, stationId, 5 as levelNumber, '
     NO2_VeryUnhealthy' as levelName insert into PollutantLevel" />
5 <Name="NO2_Unhealthy_s" text="from NO2_Avg_s[value >= 361.0 and value <
     650] select timestamp, stationId, 4 as levelNumber, 'NO2_Unhealthy' as
      levelName insert into PollutantLevel" />
6 <Name="NO2_Good_s" text="from NO2_Avg_s[value >= 0 and value < 54] select
     timestamp, stationId, 1 as levelNumber, 'NO2_Good' as levelName insert
      into PollutantLevel" />
7 <Name="AirQualityLevel_s" text="from PollutantLevel\#window.timeBatch(5
     min) select timestamp, stationId, max(levelNumber) as level group by
     stationId insert into AirQualityLevel_s" />
```

Listing 5.2: Patrones que clasifican el nivel de contaminación según la concentración en Siddhi.

5.1.4. Configuración de sinks

Una vez definidos los patrones y sus streams de salida correspondientes, se configuraron los sinks para Esper y Siddhi asignándoles los patrones que debe escuchar cada uno. También se marcó la casilla Log Events to Disk como muestra la figura 5.4 para tener un registro de los datos y realizar el análisis con la información obtenida.



(a) Esper sink (b) Siddhi sink

Figura 5.4: Configuración de sinks

5.2. Resultados

Se ha ejecutado la configuración descrita en los apartados anteriores con las siguientes opciones: las mediciones se han realizado habilitando la opción de captura de tiempo de respuesta (RT, ver sección 6.6.6), seleccionando el modo process time (inside adapters) y escogiendo una resolución en nanosegundos.

Al finalizar la ejecución, se ha comprobado el envío de cuatrocientos cincuenta mil eventos por parte de la source y la recepción de seis millones trescientos mil eventos por parte de los sinks como se observa en la figura 5.5.

Posteriormente se han cargado las estadísticas de ejecución en el performance monitor para visualizar la tabla de estadísticas. Las figura 5.6 muestra los resultados para los dos motores. Como se puede observar, el rendimiento promedio general de Esper para este ejemplo ha sido mejor que el de Siddhi, sin embargo el mejor rendimiendo máximo lo presentó Siddhi.

El sistema cuenta con una interfaz gráfica para visualización de estadísticas, como se muestra por ejemplo en la figura 5.7 en la que se indica la relación entre el evento `NO2_-Unhealthy` y el rendimiento promedio. Además, también se pueden tomar los datos del registro generado por el performance monitor para elaborar nuestros propios informes como se puede ver en la figura 5.8. Esta gráfica, que ha sido generada manualmente en una hoja

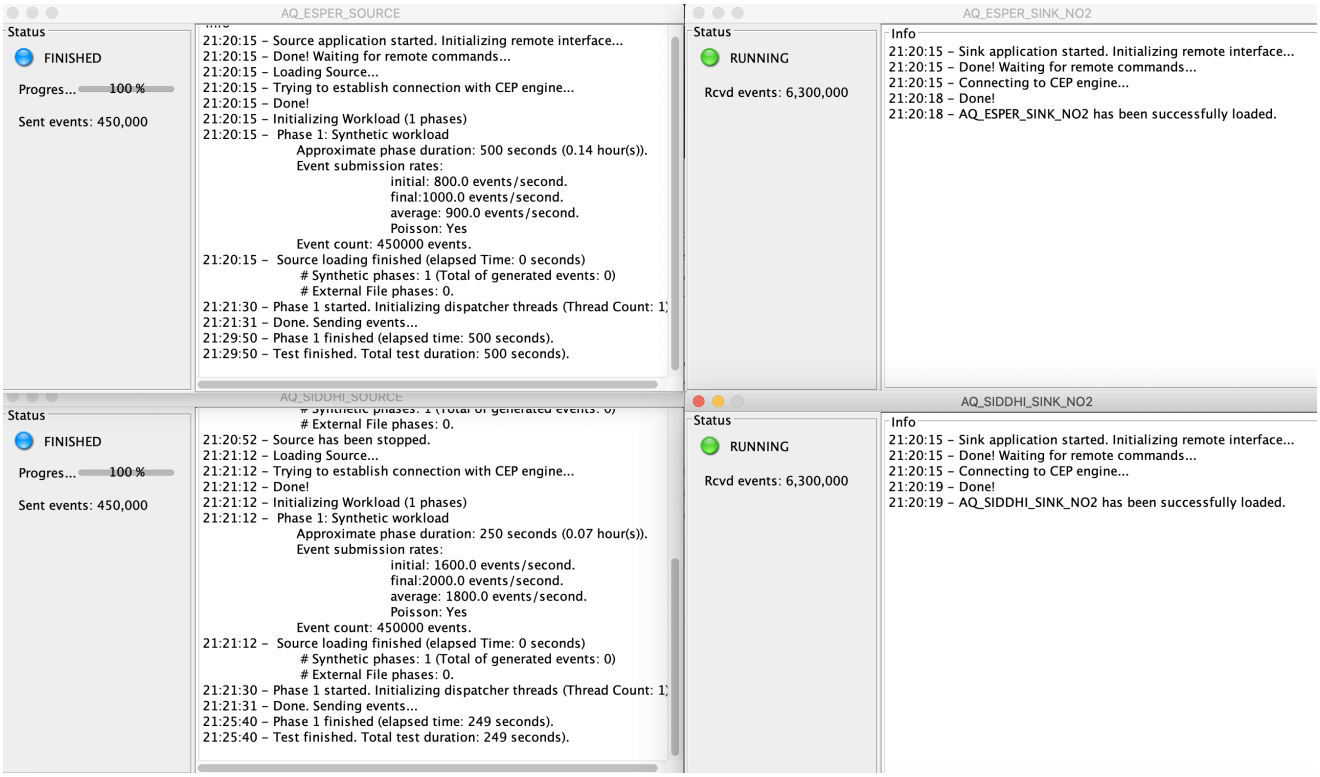


Figura 5.5: Envío y recepción de eventos sobre Esper y Siddhi

de cálculo a partir de los datos en formato CSV entregados por el sistema, compara el rendimiento promedio de ambos motores CEP para el patrón `AirQualityLevel`. Como se puede observar, el framework FINCoS++ permite una comparación de múltiples motores CEP de forma simultánea. Debe tenerse en cuenta que para este caso de estudio se han ejecutado ambos motores en la misma máquina, lo que puede producir interferencias en el rendimiento mostrado en los resultados, pero ilustra claramente las posibilidades del framework desarrollado.

Esper (sample)

Query	Avg Throughput	Max Throughput	Max RT (ms)
AirQualityLevel	15759.48	620372.00	39848318.780
NO2_Unhealthy	2936.93	119966.00	0.000
NO2_Good	0.14	26.00	0.000
NO2_Avg	4207.77	168602.00	0.000
NO2_UnhealthyForSensitiveGroup	5462.37	214490.00	0.000
NO2_Moderate	22.84	2884.00	0.000

Siddhi (Sample)

Query	Avg Throughput	Max Throughput	Max RT (ms)
AirQualityLevel	13063.48	681258.00	40028132.201
NO2_Unhealthy	2024.32	119110.00	0.000
NO2_Good	0.10	26.00	0.000
NO2_Avg	3309.20	167350.00	0.000
NO2_UnhealthyForSensitiveGroup	4583.85	213130.00	0.000
NO2_Moderate	15.24	2844.00	0.000

Figura 5.6: Tabla de estadísticas Esper vs Siddhi

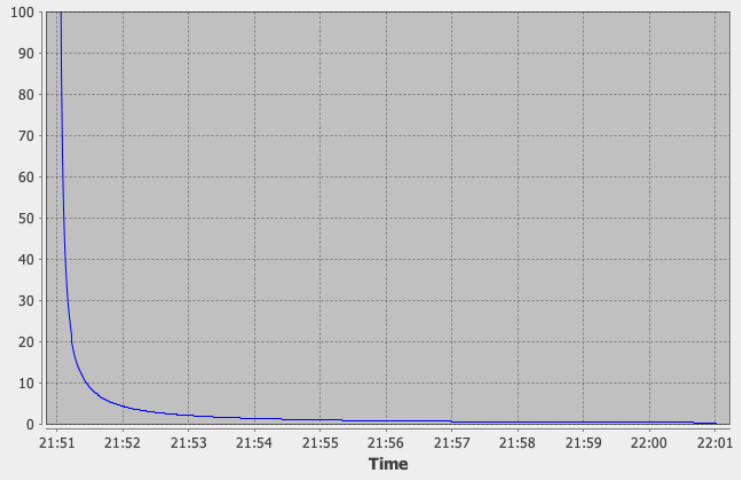


Figura 5.7: No2_Unhealthy vs Avg_Throughput generado por el sistema

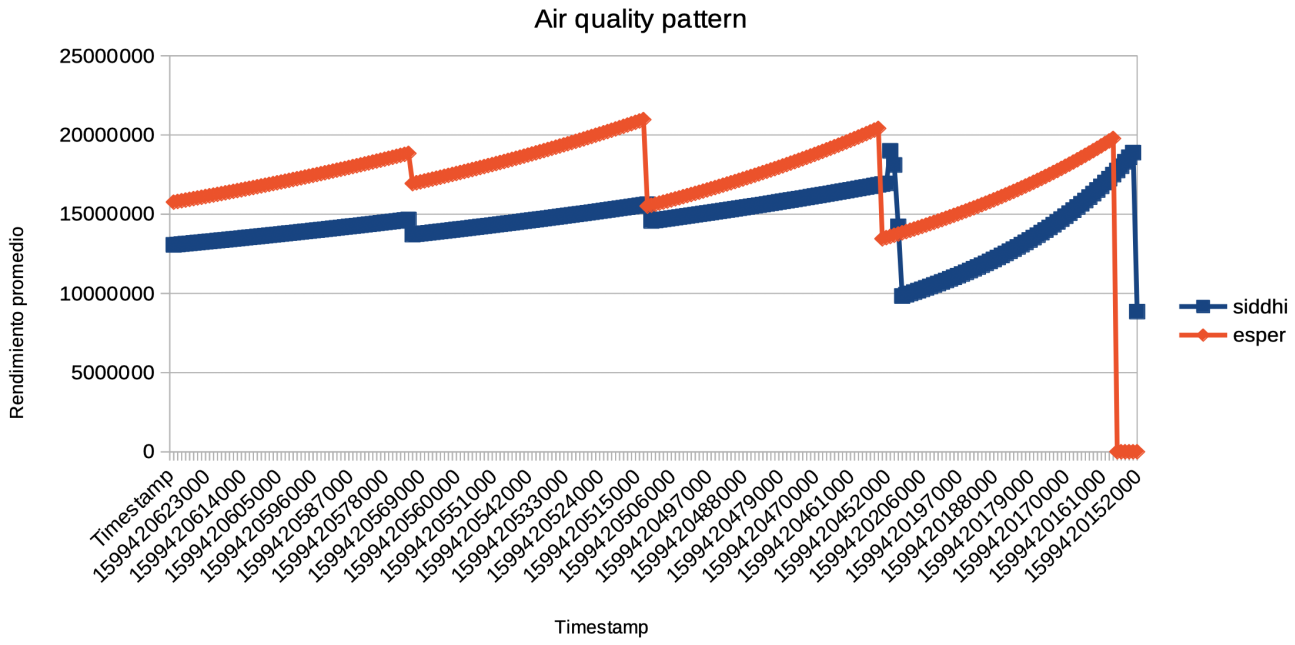


Figura 5.8: Comparación del rendimiendo promedio para el patrón de calidad del aire

CAPÍTULO 6

MANUAL DE USUARIO

6.1. Licencias asociadas a FINCoS++

Al igual que Esper y FINCOS, FINCoS++ está bajo los términos de GNU (General Public License), Versión 2 [41]. Adicionalmente, el uso de Siddhi opera bajo los términos de la licencia Apache, Versión 2.0 [15].

6.2. Repositorio asociado al trabajo

El código abierto relacionado con la aplicación desarrollada se puede encontrar en la plataforma GitHub, asociada al siguiente repositorio: https://github.com/AlexT161/TFM_FINCoS.git

6.3. Mejoras realizadas en FINCoS++ sobre la versión anterior

FINCoS++ incluye las siguientes diferencias y mejoras comparado con la versión 2.4.2 de FINCoS (Copyright (C) 2013 CISUC, University of Coimbra) [36] [38]:

- Cambio de denominación: Los drivers de FINCoS ahora se llaman sources.

- Cambio de denominación: Las queries de FINCoS ahora se llaman patterns.
- Los stream schemas se crean desde la interfaz de usuario, anteriormente se debían modificar los archivos .xml manualmente.
- Los patrones se crean desde la interfaz de usuario, anteriormente se debían modificar los archivos .xml manualmente.
- Al crear streams, patterns, sources y sinks, se puede escoger el motor hacia el cual van dirigidos los datos. En FINCoS no era necesario porque solo se utilizaba Esper y no estaban habilitadas los menús de streams y patterns.
- En la configuración de los sinks, aparece una lista desplegable con los streams disponibles para cada motor. En FINCoS era necesario editar el archivo `Stream_Set.xml` para introducir los streams disponibles para cada motor CEP y cualquier error tipográfico generaba fallos durante la ejecución.
- Se ha adaptado el sistema para utilizar la última versión de Esper, 8.4.
- Se ha añadido *Siddhi versión 5.1.2* a los motores suscritos.
- Se ha añadido la posibilidad de generar para una misma conexión, dos o más ejecuciones de Esper simultáneamente.
- Se ha ampliado el sistema bajo pruebas de un motor a múltiples motores.
- Se ha incluido la posibilidad de realizar pruebas en varios motores al tiempo, lo cual permite una comparación directa entre ellos.
- Se ha añadido una opción para acceder al offline performance monitor desde el menú principal de FINCoS++.

En la Sección 4.1 se describen los 6 componentes principales que integran la arquitectura de FINCoS++: Controller GUI, sources, sinks, adapters, performance monitor y daemon service.

6.4. Librerías externas

- Esper (v.8.4): Un motor CEP de código abierto e integrable. Sus librerías están incluidas para permitir a los usuarios experimentar rápidamente con una plataforma de procesamiento de eventos reales.
- Siddhi (v.5.1): Un motor CEP de código abierto e integrable. Sus librerías están incluidas para permitir a los usuarios experimentar rápidamente con una plataforma de procesamiento de eventos reales.
- Bibliotecas de cliente JMS y HornetQ (v.2.2.14): Permiten conectarse al middleware de mensajería de código abierto HornetQ. Sus librerías se incluyen para permitir a los usuarios ejecutar pruebas con un sistema de mensajería.
- Javassist: Permite que los programas Java definan nuevas clases en tiempo de ejecución. La librería se utiliza para crear clases que corresponden a los tipos de eventos definidos durante la configuración de carga de trabajo.
- JFreeChart: Muestra gráficos en aplicaciones Java. La librería se utiliza para mostrar métricas de rendimiento en performance monitor.
- SwingX: Contiene extensiones del kit de herramientas de la GUI de *Swing*. La librería se utiliza en la interfaz de usuario de las aplicaciones controller y performance monitor.
- Además ha sido necesario añadir librerías adicionales para el funcionamiento de Siddhi 5.1 y Esper 8.4. Se puede consultar la lista completa en el repositorio de Github mencionado en el apartado 6.2.

6.5. Instalación

FINCoS++ se puede ejecutar en cualquier máquina con una máquina virtual de Java instalada (se recomienda la versión 11 de OpenJDK). Los requerimientos de hardware

dependen de las necesidades específicas para la generación de carga y la precisión en la medición de rendimiento.

A continuación se describen los pasos para la instalación de FINCoS++ en diferentes sistemas opertivos.

En Windows:

1. Asegúrese de que esté instalada la máquina virtual de Java en cualquier máquina donde se espera ejecutar FINCoS++.
2. Agregue *Java home* a las variables de entorno de Windows (click derecho en My Computer ->Properties ->Advanced –>Environment Variables ->System Variables, edite el Path agregando el directorio bin de JRE a la ruta).
3. Descomprima el archivo FINCoS++.zip (preferiblemente en C:\FINCoS++).

En Linux:

1. Asegúrese de que esté instalada la máquina virtual de Java en cualquier máquina donde se espera ejecutar FINCoS++.
2. Descomprima el archivo FINCoS++.zip en el directorio de su preferencia.

En Mac OS:

1. Asegúrese de que esté instalada la máquina virtual de Java en cualquier máquina donde se espera ejecutar FINCoS++.
2. Descomprima el archivo FINCoS++.zip en el directorio de su preferencia.

6.6. Configuración de pruebas

La configuración de FINCoS++ se realiza a través del controller GUI. Para crear una prueba se debe configurar al menos un stream schema, una source, un pattern y un sink.

Las configuraciones de prueba pueden ser salvadas en archivos XML de configuración y pueden ser reutilizadas.

6.6.1. Configuración de schemas

Para agregar un nuevo schema seleccione New... dentro del menú Schemas, se desplegará una ventana como la ilustrada en la Figura 6.1 con los siguientes parámetros:

- Event Type: El nombre del evento a configurar.
- CEP Engine: El motor CEP al cual se enviarán los eventos que se están configurando, se puede escoger entre Esper y Siddhi.
- Property Name: El nombre de los atributos propios del evento.
- Data Type: El tipo de dato de origen de cada propiedad, se puede escoger entre `boolean`, `double`, `float`, `integer`, `long` y `text`.
- Properties: Una lista donde se van agregando o eliminando las propiedades con sus tipos de dato del evento a configurar.

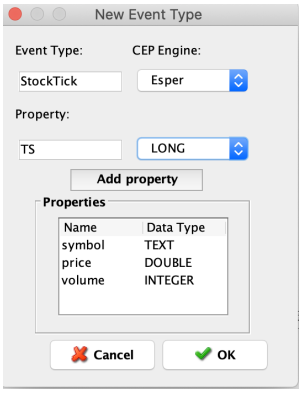


Figura 6.1: Ventana de configuración de schemas

Una vez terminada la configuración del esquema, se debe pulsar el botón OK para que se almacene el esquema dentro del archivo stream set correspondiente al motor escogido. FINCoS++ validará que todos los campos hayan sido rellenados.

6.6.2. Configuración de sources

Para agregar una nueva source seleccione New... dentro del menú Sources, se desplegará una ventana como la ilustrada en la Figura 6.2 con los siguientes parámetros:

- Alias: Un identificador único de la source en esa configuración de prueba.
- Address: La dirección IP de la máquina donde se ejecutará la source.
- Send Events to: Una conexión con un motor CEP o proveedor JMS a través del cual se enviarán los eventos.
- Workload: Un conjunto de fases, cada una con sus propias características de carga. Cada source debe tener al menos una fase.
- Thread Count: Establece el número de threads utilizados para la generación de carga/envío de eventos.
- Logging: Especifica si los eventos generados se registran en el disco o no. También configura si se deben almacenar todos los atributos de los eventos o solo sus timestamps, si se realiza o no muestreo, y la frecuencia con la que las entradas del registro se almacenan en el disco (Flush Interval).

Todos los campos son obligatorios y deben ser rellenados correctamente. La carga de trabajo de una source se compone de fases, que pueden ser sintéticas (es decir, generadas por FINCoS++), o basadas en un archivo externo proporcionado por el usuario.

6.6.3. Configuración de patterns

Para agregar un nuevo pattern seleccione New... dentro del menú Patterns, se desplegará una ventana como la ilustrada en la Figura 6.3 con los siguientes parámetros:

- Pattern Name: El nombre del patrón a crear.

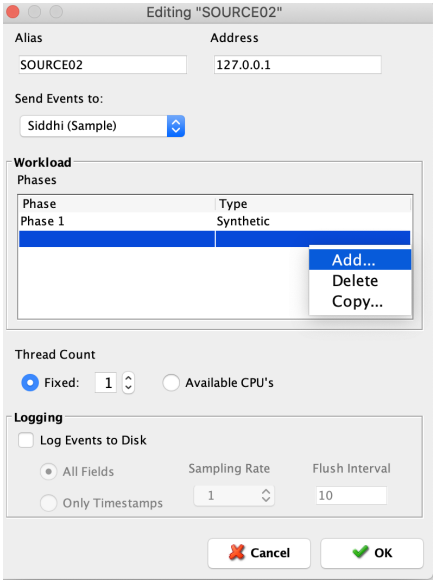


Figura 6.2: Ventana de configuración de sources

- CEP Engine: El motor CEP sobre el cual se ejecutará el patrón creado.
- Pattern: El patrón en la sintaxis del motor escogido (es importante que el usuario domine la sintaxis del motor que desea poner bajo pruebas).

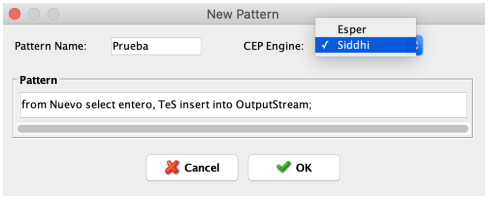
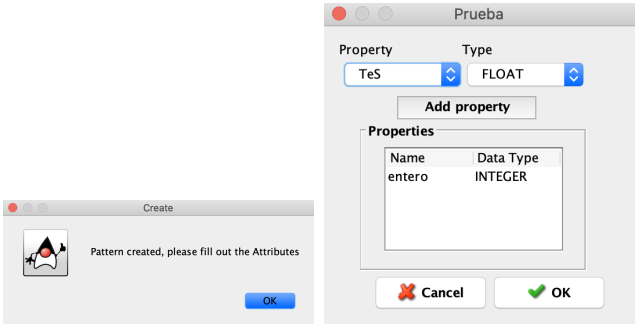


Figura 6.3: Ventana de configuración de patterns

Al pulsar OK, se desplegará un cuadro de diálogo pidiendo que acepte almacenar el patrón en el query set correspondiente al motor escogido (Figura 6.4a).

Una vez realizado este paso, se desplegará una nueva ventana en la cual se podrá escoger el tipo de dato que le corresponde a cada atributo del patrón almacenado o editado (Figura 6.4b), este schema se almacenará como *output stream* en el stream set correspondiente al motor escogido.



(a) Crear schema (b) Editar schema

Figura 6.4: Editor de pattern schemas

6.6.4. Configuración de sinks

Para agregar un nuevo sink seleccione New... dentro del menú Sinks, se desplegará una ventana como la ilustrada en la Figura 6.5 con los siguientes parámetros:

- Alias: Un identificador único del sink en esa configuración de prueba.
- Address: La dirección IP de la máquina donde se ejecutará el sink.
- Receive Events from: Una conexión con un motor CEP o proveedor JMS del cual se recibirán los eventos.
- Streams: La lista de output streams que el sink escuchará. Se pueden agregar y borrar output streams usando el click derecho como lo muestra la Figura 6.5 y escogiendo el stream de una lista desplegable (se debe crear al menos un pattern antes de configurar el sink).
- Logging: Especifica si los eventos generados se registran en el disco o no. También configura si se deben almacenar todos los atributos de los eventos o solo sus timestamps, si se realiza o no muestreo, y la frecuencia con la que las entradas del registro se almacenan en el disco (Flush Interval).

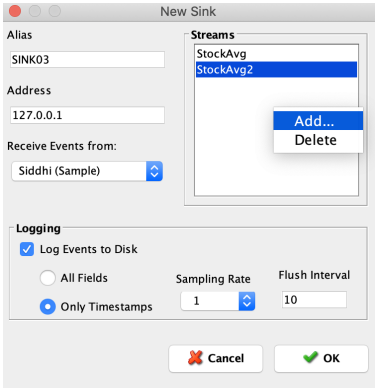


Figura 6.5: Ventana de configuración de sinks

6.6.5. Conexiones

FINCoS++ intercambia eventos con motores CEP a través de conexiones preconfiguradas. Para crear una nueva conexión o editar alguna existente, seleccione Connections en el menú View, se desplegará una ventana como la mostrada en la figura 6.6.

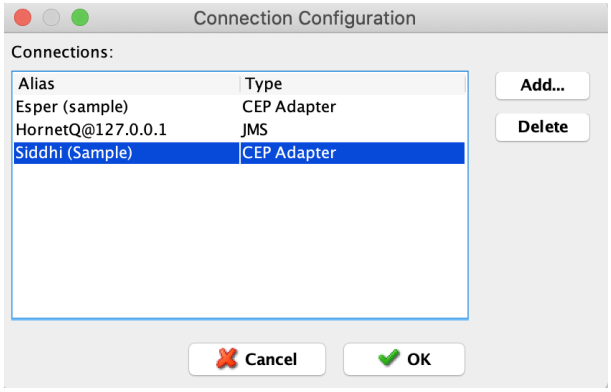


Figura 6.6: Ventana Connection Configuration

Las conexiones configuradas son almacenadas en un archivo, y se puede utilizar en diferentes configuraciones de prueba. Para configurar una conexión nueva, de click en el botón Add...; para editar una existente, doble click en ella. Aparecerá una ventana como la mostrada en la Figura 6.7.

Hay dos tipos de conexiones: CEP Adapter y *JMS (Java Message Service)*. Con el primero, las sources y los sinks se conectan directamente a un motor CEP utilizando su API;

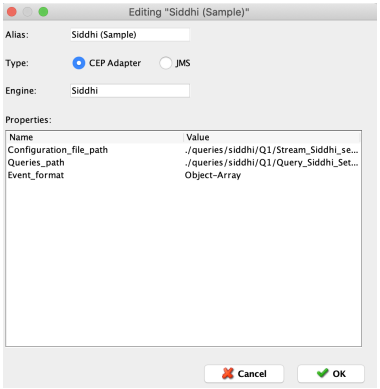


Figura 6.7: Edición de una conexión existente

mientras que con el segundo, los eventos se intercambian con el motor CEP utilizando un middleware de mensajería compatible con JMS como intermediario. Tenga en cuenta que la primera opción requiere codificar un adaptador personalizado para cada producto que se pretende probar (actualmente FINCoS++ cuenta con adaptadores para los motores de código abierto Esper y Siddhi). Si en su lugar se utiliza el adaptador JMS, es necesario configurar una cola para cada tipo de evento enviado en las pruebas.

Una conexión tiene un alias que la identifica de forma exclusiva y un conjunto de parámetros específicos del proveedor necesarios para conectarse con el motor CEP o el proveedor JMS deseado.

6.6.6. Medición del tiempo de respuesta

Seleccionando Options... en el menú Test es posible configurar si se debe medir o no el tiempo de respuesta y cómo se debe hacer (Figura 6.8).

El tiempo de respuesta se puede medir de diferentes formas y dentro de dos niveles de resolución. Las opciones Mode y Resolution definen en qué puntos y en qué momento se recopilan los timestamp de resolución para el cálculo del tiempo de respuesta.

La opción Mode permite escoger entre *end-to-end*, donde el tiempo de respuesta se mide como el tiempo que tarda un evento de salida en llegar a un sink después de que una source envía el evento correspondiente que lo causó; y *Process Time*, donde el tiempo de respuesta

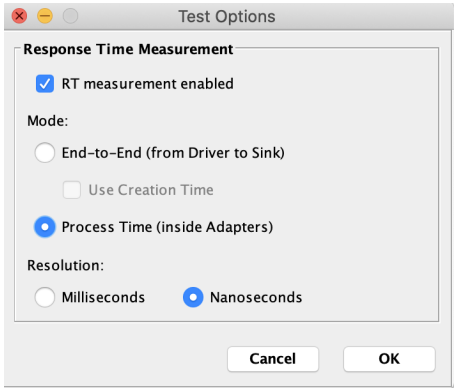


Figura 6.8: Edición de una conexión existente

se mide dentro de los adaptadores. Se calcula como la diferencia entre el momento en el que se notifica al adaptador la llegada de un evento de salida del motor CEP, y el momento en el que el evento correspondiente que causó la salida fue enviado al motor CEP por el adaptador.

La opción Resolution permite escoger si la resolución de las mediciones se realizarán en nanosegundos o en milisegundos; cabe mencionar que la resolución se puede medir en nanosegundos solo si las sources y los sinks se ejecutan en la misma máquina.

6.7. Extensión de FINCoS++ con motores CEP adicionales

Es posible realizar una evaluación de rendimiento utilizando una conexión directa con un motor CEP determinado. En ese caso, es necesario desarrollar un adaptador personalizado.

FINCoS++ contiene dos adaptadores, para Esper y Siddhi. Desarrollar un adaptador en FINCoS++ consiste en implementar dos elementos: uno para enviar eventos al motor CEP y otro para recibir eventos desde él. La conversión entre el formato FINCoS++ y la representación interna de los productos está encapsulada dentro de esas dos funciones. La clase abstracta `CEPEngineInterface` (listado 6.1) se utiliza para conectarse a los motores CEP, enviar y recibir eventos hacia/desde ellos, y recuperar la lista de streams.

```java
public abstract class CEP_EngineInterface implements InputAdapter {
    /**
     * Connects to CEP engine.
     */
    public abstract boolean connect() throws Exception;

    /**
     * Performs any vendor-specific initialization at client side
     * (e.g. initialize listeners for output streams).
     */
    public abstract boolean load(String [] outputStreams, Sink sinkInstance) throws
        Exception;

    /**
     * Disconnects from CEP engine and performs any finalization procedures needed
     * (e.g. close listeners for output streams)
     */
    public abstract void disconnect();
    public abstract void disconnect2();

    /**
     * Retrieves the list of input streams on the CEP engine
     * (optional function).
     */
    public abstract String[] getInputStreamList() throws Exception;


    /**
     * Retrieves the list of output streams on the CEP engine
     * (optional function).
     */
    public abstract String[] getOutputStreamList() throws Exception;
}
```

Listing 6.1: CEPEngineInterface

Para extender la aplicación del adaptador para que admita otros motores CEP, se debe crear una clase que herede de la clase `CEPEngineInterface` e implementar sus métodos abstractos, como se muestra en la sección 4.3.2. También se necesita implementar un listener para recibir los output streams del motor CEP. Se pueden encontrar más detalles en el código fuente de FINCoS++.

CAPÍTULO 7

CONCLUSIONES

Una vez explicada la arquitectura, la operación y la configuración de FINCoS++, y realizadas las pruebas de funcionamiento, en este capítulo presentamos las conclusiones y posibles extensiones a desarrollar en un futuro.

7.1. Conclusiones

Como se esperaba al inicio de este proyecto, se consiguió actualizar y extender una herramienta de análisis de rendimiento de motores CEP que evalúa sus tiempos de procesamiento con un nivel de granularidad que no proporciona ninguna otra aplicación.

Esto ha sido posible gracias al uso de diferentes herramientas, tecnologías y técnicas de programación como lo son la programación orientada a objetos en lenguaje Java, la programación concurrente soportada en threads y swingworkers, la comunicación remota mediante la interfaz RMI de Java, el uso de patrones de diseño como facade y un entorno gráfico basado en Swing. También se han utilizado los propios lenguajes de los motores CEP: EPL de Esper y Siddhi Streaming SQL de Siddhi, para el desarrollo de patrones y streams en dichos motores, pues realizar las pruebas de desarrollo sin dicho conocimiento resultaría más complicado o podría inducir a conclusiones erróneas.

La herramienta ahora funciona utilizando la versión 8.4 de Esper, lo cual permite la

ejecución de múltiples pruebas con una misma instancia del motor, y se ha integrado un adaptador para trabajar con un motor adicional, Siddhi versión 5.1.

Adicionalmente, las mejoras funcionales respecto a la automatización del sistema, la creación de nuevos menús, la integración del informe de estadísticas y el ajuste de definiciones fueron implementadas satisfactoriamente.

En cuanto a las pruebas de rendimiento sobre los motores, observamos que el comportamiento de la herramienta es estable cuando se ejecutan pruebas usando 2 motores simultáneamente y que al final de la ejecución es capaz de entregar un informe de estadísticas midiendo el rendimiento y el tiempo de respuesta de los motores CEP.

Como dato adicional, observamos que para las pruebas realizadas, el motor Esper presentó un mejor rendimiento promedio ejecutándose en simultáneo. Podemos concluir que este trabajo permite a FINCoS ++ realizar comparaciones simultáneas entre diferentes motores CEP.

7.2. Extensiones y trabajo futuro

Como se menciona en la sección 6.7, es posible realizar extensiones para trabajar con motores CEP adicionales. Se puede iniciar su extensión revisando los motores vigentes descritos en las secciones 2 y 3.

Desafortunadamente, cada motor CEP es un sistema complejo con sus propias características que impiden generalizar y automatizar las adaptaciones realizadas en este trabajo para otros sistemas además de Siddhi. Sin embargo, este trabajo puede utilizarse para identificar los componentes que deben modificarse para extender FINCoS++ a otros sistemas.

Otra útil y no menos importante ampliación que se puede realizar consiste en incluir una interfaz web a la nueva versión de FINCoS++. Esto permitiría tener usuarios conectados con la capacidad de compartir diseños de sources y sinks, ampliando aún más el campo de acción de la aplicación.

Además de esto, una mejora funcional muy interesante en sistemas multiplataforma como

FINCoS++ consiste en la integración de un sistema de traducción automática de lenguajes de procesamiento de eventos. Esto puede proporcionar aún más posibilidades al usuario que no tenga conocimientos en el manejo de distintos motores de procesamiento de eventos; así por ejemplo, el usuario podría efectuar pruebas de rendimiento sobre un motor como Siddhi a partir de sus patrones y esquemas previamente diseñados para ejecutarse en Esper.

CAPÍTULO 8

INTRODUCTION

> "Making the simple complicated is commonplace; making the complicated simple, awesomely simple, that's creativity."
>
> *Charles Mingus*

8.1. Motivation

In today's world, an exorbitant amount of data comes from different sources such as sensors, transactions, web activity, and social networks, among others. The ability to accumulate, analyze, and react to this stream of events in real time is becoming a key component of information systems, both in business and in the field of research. Making sense of this data overflow requires fast and efficient processing.

Processing a large volume of events that are transformed into valuable information is a vital part of decision making, and complex event processing has become one of the fastest-growing emerging fields as both a research discipline and industrial trend.

However, we still lack information about the performance of complex event processing engines, considering that standard benchmarks were not available until FINCoS appeared. The main FINCoS's goal was to fill this gap by providing a flexible and neutral approach

through which users could quickly run realistic performance tests on one or more event processing engines without having to personally configure event conversion and load phases [36].

FINCoS development began in 2007, as part of the BiCEP project [11]. At the time, there was not no much information on the use of complex event processing systems and a wide variety of products lived together, each with their own languages and styles, posing significant challenges for development of new frameworks for their performance analysis and comparison. The first version of FINCoS was released in 2008, and its latest update was made in 2013. Since then, no new FINCoS updates have been released and the version that is available includes just one engine adapter to Esper.

To continue with this work, we propose to generate or improve a tool that meets these objectives and that also allows working with different event processing engines. To carry out this proposal, we have seen necessary to download FINCoS, update its version, make improvements to its performance, optimize its functionality, and extend it to more complex event processing engines for simultaneous performance testing.

The challenge posed involves analyzing a system of the complexity of FINCoS without having more material than its source code to identify its operation in depth and face the technologies it uses while developing the new version.

8.2. Objectives

The general main objective of this work is to expand and improve a framework for load generation and for testing of CEP engines performance in a flexible and neutral way. This tool will provide users with a system that uses the latest versions of CEP engines and generate artificial workloads as well as to use real data sets, allowing to evaluate different candidate solutions. At the same time, developers of engines for processing complex events can compare and improve their performance.

The specific objectives of this work are as follows:

- Update FINCoS to be able to use the latest version of Esper, 8.4. In order to do that, the differences with version 4.9 must be identified and the FINCoS components that perform bidirectional communication with Esper must be modified.
- Analyze more CEP engines to extend the FINCoS system with another of the most used engines.
- Make functional improvements to maintain consistency of complex event data configured for FINCoS.
- Test the new system with some case studies.

8.3. Workplan

To achieve the fulfillment of the main objective, it is necessary to carry out an effective management of the project and divide it into functional milestones with their respective tasks. The following list shows the main milestones and tasks to be achieved throughout this work:

1. Familiarization with FINCoS
    a) Thorough review of the available documentation.
    b) Installation and tests on the current version.
    c) In-depth study of the source code.
2. Migration to Esper 8.4.
    a) Thorough review of the Esper documentation.
    b) Analyze the FINCoS components that communicate with Esper and modify them to adapt to version 8.4.
    c) Test new configuration with Esper 8.4.

3. Integration of new engines.

    a) Review additional CEP engines documentation.

    b) Select CEP engines to integrate.

    c) Perform integration of new engines to the system.

    d) Test the operation of the integration.

4. Functional improvements.

    a) Automate the system.

    b) Create new menus.

    c) Integrate statistics report.

    d) Adjust definitions.

5. Comparison tests with integrated engines.

8.4. Document structure

The structure of the document is as follows:

- In the chapter 2 are the essential definitions to understand the work developed about the processing of complex events.
- Chapter 3 reviews the main work related to performance comparison of these engines and how far progress has been made in this field.
- Chapter 4 describes the architecture of FINCoS ++, its class diagram, the programming techniques used to optimize its operation, and the improvements included in this new version, as well as the steps taken to reach these improvements.
- Chapter 5 presents the tests and results performed in FINCoS ++ to illustrate how it works.

- Chapter 6 includes the system user manual and its operating requirements.
- Chapter 7 shows the conclusions and possible future extensions on FINCoS ++.
- Appendix A, class diagram of FINCoS ++

CAPÍTULO 9

CONCLUSIONS

Once the architecture, operation, and configuration of FINCoS++ have been explained and the functional tests have been carried out, in this chapter we present the main conclusions to this work and possible extensions to be developed in the future.

9.1. Conclusions

As expected at the beginning of this project, it was possible to update and extend a CEP engine performance analysis tool that evaluates its processing times with a level of granularity that no other application provides.

This has been possible thanks to the use of different tools, technologies and programming techniques such as object-oriented programming in Java language, concurrent programming librarires supported by threads and swingworkers, remote communication through the Java RMI interface, the use of design patterns as facade and a graphical environment based on Swing. The CEP engines' own languages have also been used: EPL for Esper and Siddhi Streaming SQL for Siddhi, for the development of patterns and streams in these engines, as carrying out development tests without such knowledge would have been more complicated or could have even led to wrong conclusions.

The tool now works using Esper version 8.4, which allows the execution of multiple tests

with the same instance of the engine, and one adapter has been integrated to work with an additional engine, Siddhi version 5.1.

Additionally, functional improvements regarding the automation of the system, the creation of new menus, the integration of the statistics report and the adjustment of definitions have been implemented successfully.

Regarding the performance tests on the engines, we have observed that the behavior of the tool is stable when running tests with 2 engines simultaneously and that at the end of the execution it is capable of delivering a statistics report measuring the performance and the time of response of CEP engines.

As additional data, we have observed that, for the tests carried out, the Esper engine presented the best average performance running simultaneously. We can conclude that this work enables FINCoS++ for making simultaneous comparisons between different CEP engines.

9.2. Extensions and future work

As mentioned in section 6.7, it is possible to make extensions to this work with additional CEP engines. These extensions can be started by reviewing the current engines described in sections 2 and 3. Unfortunately, each CEP engine is a complex system with its own characteristics that prevent generalizing and automating the adaptations made in this work for systems other than Siddhi. However, this work can be used to identify the components that must be modified to extend FINCoS ++ to other systems.

Another useful and no less important extension that can be made is to include a web interface to the new version of FINCoS++. This would allow connected users with the ability to share sources and sinks designs, further expanding the scope of the application.

In addition to this, during the development of this project we have seen that a very interesting functional improvement in multiplatform systems such as FINCoS++ consists of the integration of an automatic translation system for event processing languages. This

could provide even more possibilities for the user who does not have knowledge of different event processing engines; for example, the user will be able to perform performance tests on an engine such as Siddhi from its patterns and schemas previously designed to run in Esper.

BIBLIOGRAFÍA

[1] 2020-Oracle. Java se 6 downloads. https://www.oracle.com/java/technologies/javase-java-archive-javase6-downloads.html.

[2] 2020-Oracle-Corporation. Openjdk 11. https://openjdk.java.net/projects/jdk/11/.

[3] Paschke Adrian. Design patterns for complex event processing. *Technical University Dresden*, 2008.

[4] Buchmann Alejandro and Koldehofe Boris. Complex event processing. *Universität Stuttgart*, 5:241–242, 2009.

[5] D. Alur, NJ. Crupi, and D.B. Malls. *Core J2EE Patterns: Best Practices and Design Strategies*, volume 2nd Ed. New York: Prentice-Hall/Sun-Press, 2001.

[6] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A high-performance event processing engine, 2007.

[7] Fernando Chiran. Building an efficient ci/cd pipeline for siddhi. https://medium.com/

siddhi-io/building-an-efficient-ci-cd-pipeline-for-siddhi-c33150721b5d, 8 2019.

[8] D. Corral-Plaza, J. Boubeta-Puig, G. Ortiz, and A. García de Prado. An internet of things platform for air station remote sensing and smart monitoring. *Comput. Syst. Sci. Eng.*, 35(1), 2020.

[9] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: from data stream to complex event processing, 2011.

[10] C. Luckham David. *Event Processing for Business : Organizing the Real-Time Enterprise*, volume 1st Ed. New Jersey: John Wiley & Sons, Incorporated, 2011.

[11] Facultade de ciências e tecnologia da Universidade de Coimbra. Bicep proyect. http://bicep.dei.uc.pt, 2007.

[12] A. García de Prado, G. Ortiz, J. Boubeta-Puig, and D. Corral-Plaza. Air4people: a smart air quality monitoring and context-aware notification system. *J. UCS*, 24(7):846–863, 2018.

[13] Yanlei Diao, Neil Immerman, Haopeng Zhang, Daniel Gyllstrom, Jagrati Agrawaln, and Eugene Wu. Sase. http://avid.cs.umass.edu/sase/index.php?page=templates_and_stylesheets, 2014 (Retrieved on August 9 2020).

[14] Eclipse-Foundation-Inc. Eclipse ide for java developers. https://www.eclipse.org/downloads/packages/release/kepler/sr1/eclipse-ide-java-developers.

[15] The Apache Software Foundation. Apache license, version 2.0. http://www.apache.org/licenses/LICENSE-2.0.

[16] The Apache Software Foundation. Anatomy of a flink program. https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/datastream_api.html, 2014-2019 (Retrieved on July 7 2020).

[17] The Apache Software Foundation. Apache flink — stateful computations over data streams. https://flink.apache.org, 2014-2019 (Retrieved on July 7 2020).

[18] The Apache Software Foundation. Apache ode. https://ode.apache.org/index.html, 2014-2019 (Retrieved on July 7 2020).

[19] The Eclipse Foundation. The viatra query language. https://www.eclipse.org/viatra/documentation/query-language.html, 2017 (Retrieved on June 5 2020).

[20] The Eclipse Foundation. Getting started with viatra. https://www.eclipse.org/viatra/documentation/tutorial.html, 2017 (Retrieved on June 7 2020).

[21] The Hillside Group. Patterns definition. https://hillside.net/patterns-definition, 1994-2018 (Retrieved on August 7 2020).

[22] Sylvain Hallé. *EVENT STREAM PROCESSING WITH BEEPBEEP 3: Log crunching and analysis made easy.* Québec: Presses de l'Université du Québec, 2019.

[23] Sylvain Hallé and Raphaël Khoury. Event stream processing with beepbeep 3. *Kalpa Publications in Computing*, 3:81–88, 2017.

[24] EsperTech Inc. *Esper Reference*, version 8.4.0 edition, 2020.

[25] EsperTech Inc. Esper reference online. chapter 2. basic concepts. http://esper.espertech.com/release-8.5.0/reference-esper/html/processingmodel.html, 2020 (Retrieved on June 8 2020.

[26] EsperTech Inc. Esper reference online. chapter 5. epl reference: Clauses. http://esper.espertech.com/release-8.5.0/reference-esper/html/epl_clauses.html#epl-select-syntax, 2020 (Retrieved on June 8 2020.

[27] Gartner Inc. Definition of complex-event processing. https://www.gartner.com/en/information-technology/glossary/complex-event-processing, 2020 (Retrieved on August 8 2020).

[28] David Istvan. Viatra/cep. https://wiki.eclipse.org/VIATRA/CEP, 2016 (Retrieved on June 6 2020).

[29] Java-Platform-SE-7. Class swingworker<t,v>. https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html.

[30] Java-Platform-SE-7. Class thread. https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html.

[31] Java-Platform-SE-7. Interface remote. https://docs.oracle.com/javase/7/docs/api/java/rmi/Remote.html.

[32] F. Naughton Jeffrey, Barman Siddharth, and He Yeye. On load shedding in complex event processing. *Cornell University*, 2013.

[33] RSA Link. Rsa netwitness platform esa. https://community.rsa.com/docs/DOC-40543.

[34] H. Macià, G. Díaz, J. Boubeta-Puig, E. Valero, and V. Valero. Combining fuzzy logic and CEP technology to improve air quality in cities. In *Computational Science - ICCS 2019 - 19th International Conference, Faro, Portugal, June 12-14, 2019, Proceedings, Part V*, volume 11540 of *Lecture Notes in Computer Science*, pages 559–565. Springer, 2019.

[35] Marcelo R.N. Mendes, Pedro Bizarro, and Paulo Marques. Assessing and optimizing microarchitectural performance of event processing system, 2010.

[36] Marcelo R.N. Mendes, Pedro Bizarro, and Paulo Marques. Fincos: Benchmark tools for event processing systems. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. CISUC, University of Coimbra, Dep. Eng. Informática, 2013.

[37] Manuel Meyer. Using complex event processing (cep) with microsoft streaminsight to analyze twitter tweets 2: What are cep and streaminsight? https://www.12qw.ch/2013/10/streaminsight-cep-2-what-are-cep-and-streaminsight/, 10 2013.

[38] Mendes M.R.N., Bizarro P., and Marques P. Fincos. https://code.google.com/archive/p/fincos/.

[39] Siddam Nagarjuna. Consistent splitting of event streams in parallel complex event processing. Master's thesis, University of Stuttgart, 2015.

[40] Raghunath Nambiar. Second tpc technology conference, tpctc 2010. In *Performance Evaluation, Measurement and Characterization of Complex Systems*. Springer, 2010.

[41] Opensource.org. Gnu general public license version 2. https://opensource.org/licenses/gpl-2.0.php.

[42] Oracle. *Oracle Complex Event Processing EPL Reference Guide.*, 2008.

[43] Nicolás Passerini, Javier Fernandes, and Esteban Lorenzano. Dsl - xtext. https://sites.google.com/site/programacionhm/conceptos/dsls/domainspecificlanguage/dsl---xtext, 2017 (Retrieved on June 6 2020).

[44] C. Imthyaz Sheriff and Angelina Geetha. A comprehensive framework for complex event processing. *International Journal Of Recent Advances in Engineering & Technology (IJRAET)*, 1, 2013.

[45] Siddhi and WSO2. Siddhi 5.1 architecture. https://siddhi.io/en/v5.1/development/architecture/, 2019 (Retrieved on July 10 2020).

[46] Siddhi and WSO2. Siddhi 5.1 documentation. https://siddhi.io/en/v5.1/docs/, 2019 (Retrieved on July 10 2020).

[47] Siddhi and WSO2. Siddhi 5.1 streaming sql guide. https://siddhi.io/en/v5.1/docs/query-guide/, 2019 (Retrieved on June 20 2020).

[48] Siddhi and WSO2. Siddhi application. https://siddhi.io/en/v5.1/docs/examples/siddhiapp/, 2019 (Retrieved on June 20 2020).

[49] Siddhi and WSO2. *Siddhi 5.1 Quick Start Guide*, version 5.1 edition, 2019 (Retrieved on June 8 2020).

[50] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Srinath Perera, Subash Chaturanga, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures, 2016.

[51] The-Apache-Software-Foundation. What is apache flink? — architecture. https://flink.apache.org/flink-architecture.html, 2014-2019 (Retrieved on July 7 2020).

[52] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *ACM SIGMOD*, pages 407–418, 2006.

APÉNDICE A

DIAGRAMA DE CLASES

Para poder visualizar de forma legible el diagrama de clases global del proyecto, se han tomado algunos fragmentos de las clases más importantes involucradas en este; estas se presentan a continuación.

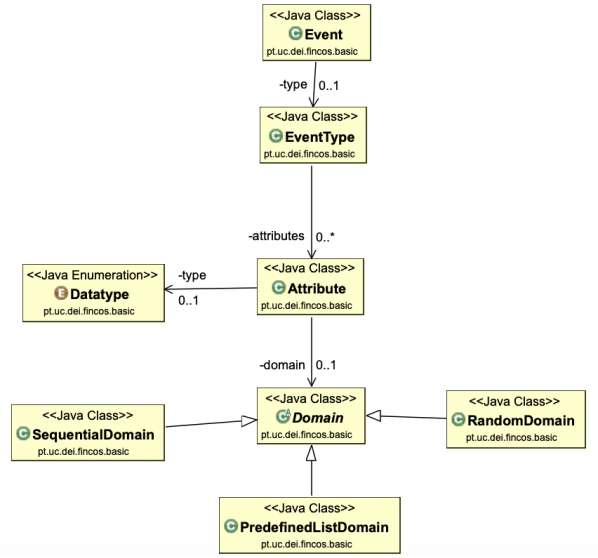


Figura A.1: Diagrama de clases basic

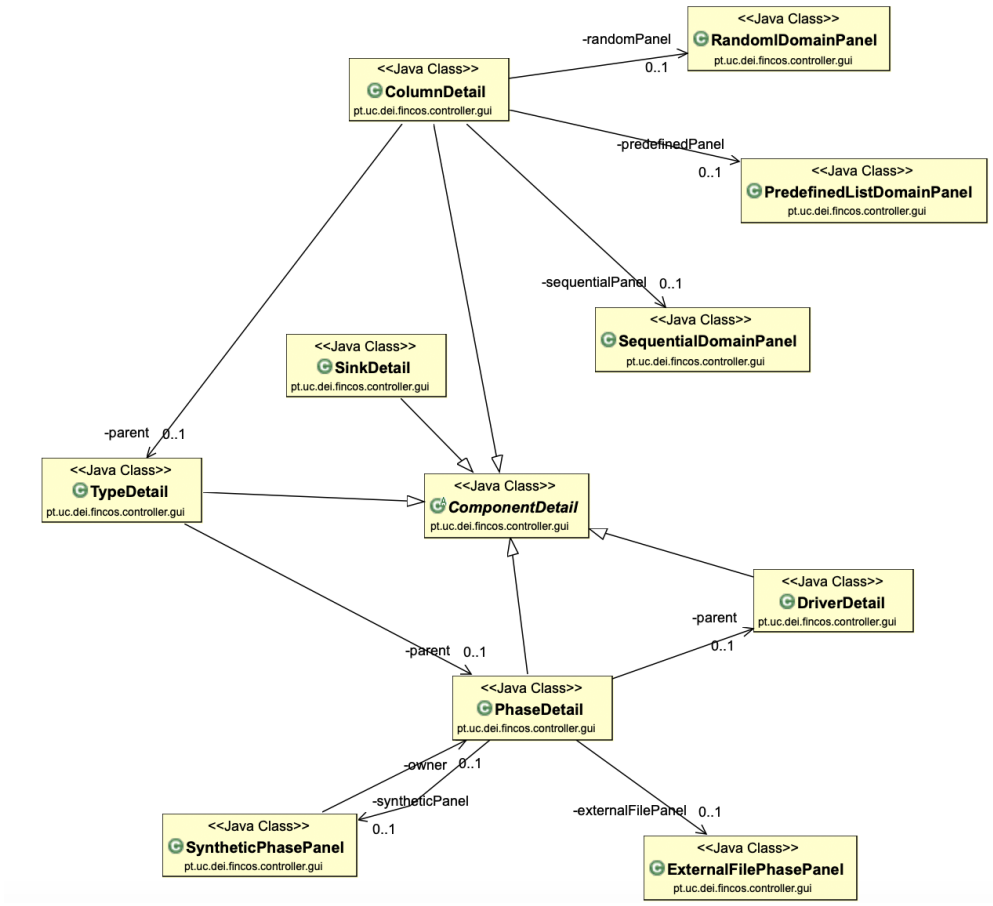


Figura A.2: Diagrama de clases component detail 1

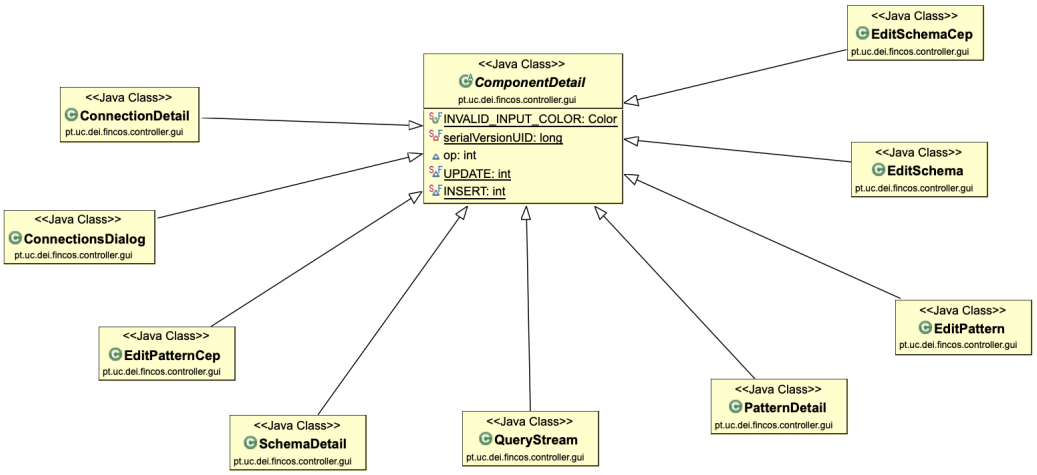


Figura A.3: Diagrama de clases component detail 2

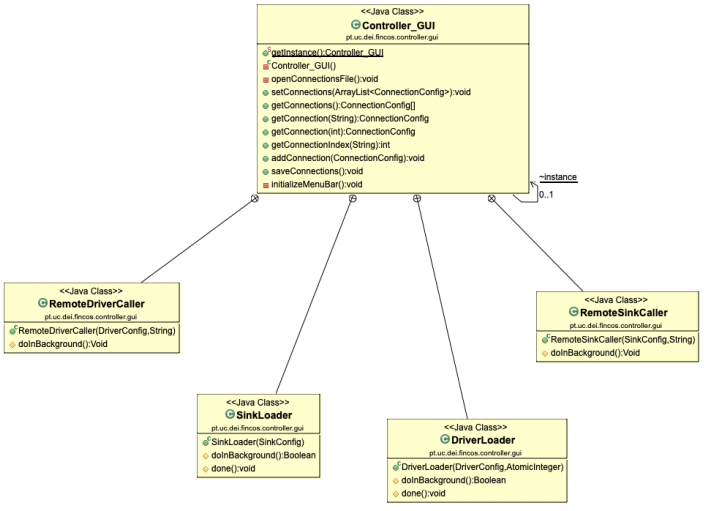


Figura A.4: Diagrama de clases Controller_GUI

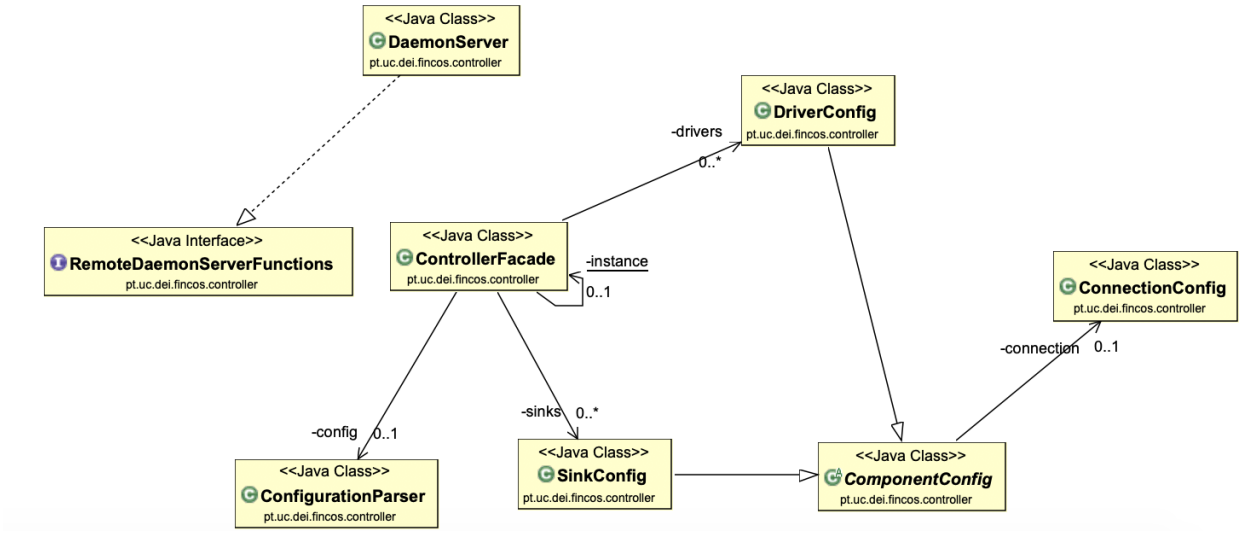


Figura A.5: Diagrama de clases controller facade

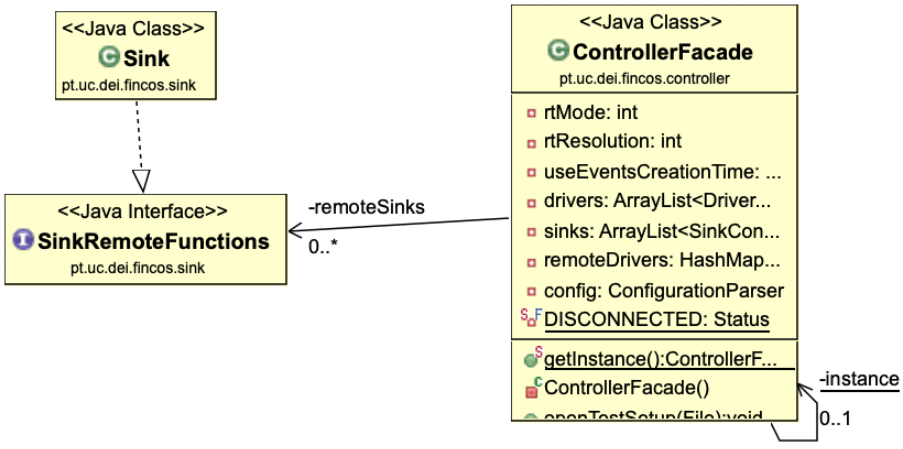


Figura A.6: Diagrama de clases sink

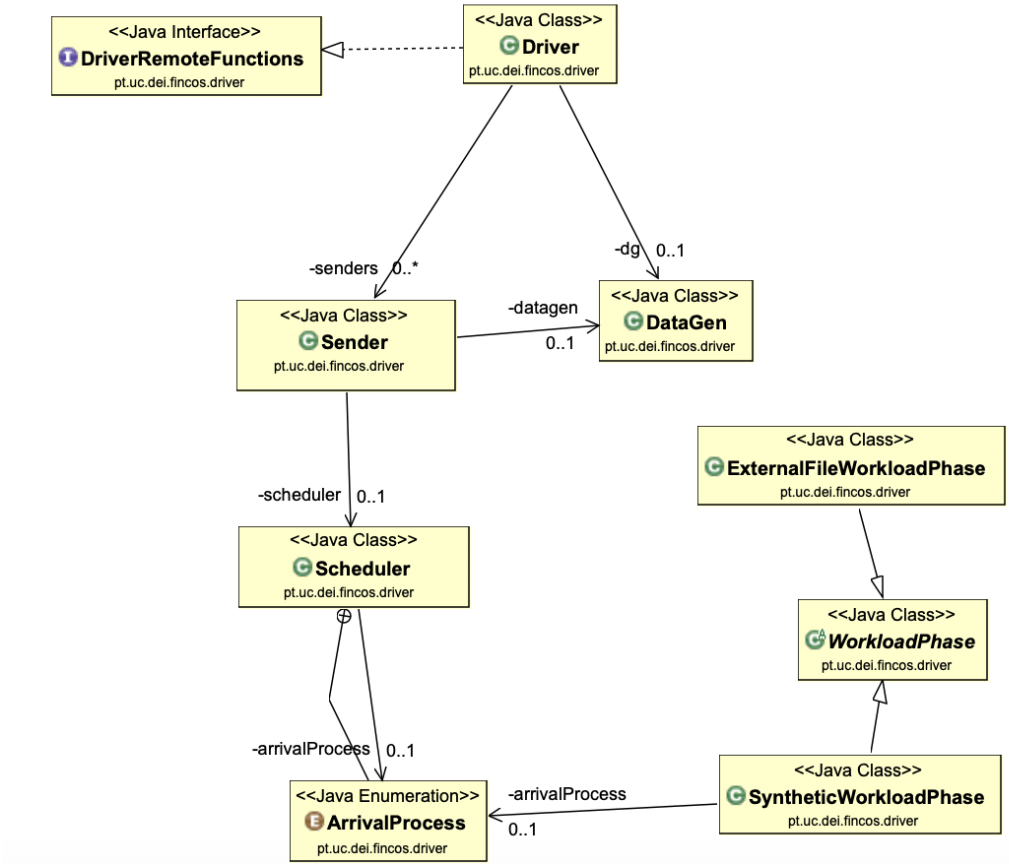


Figura A.7: Diagrama de clases driver

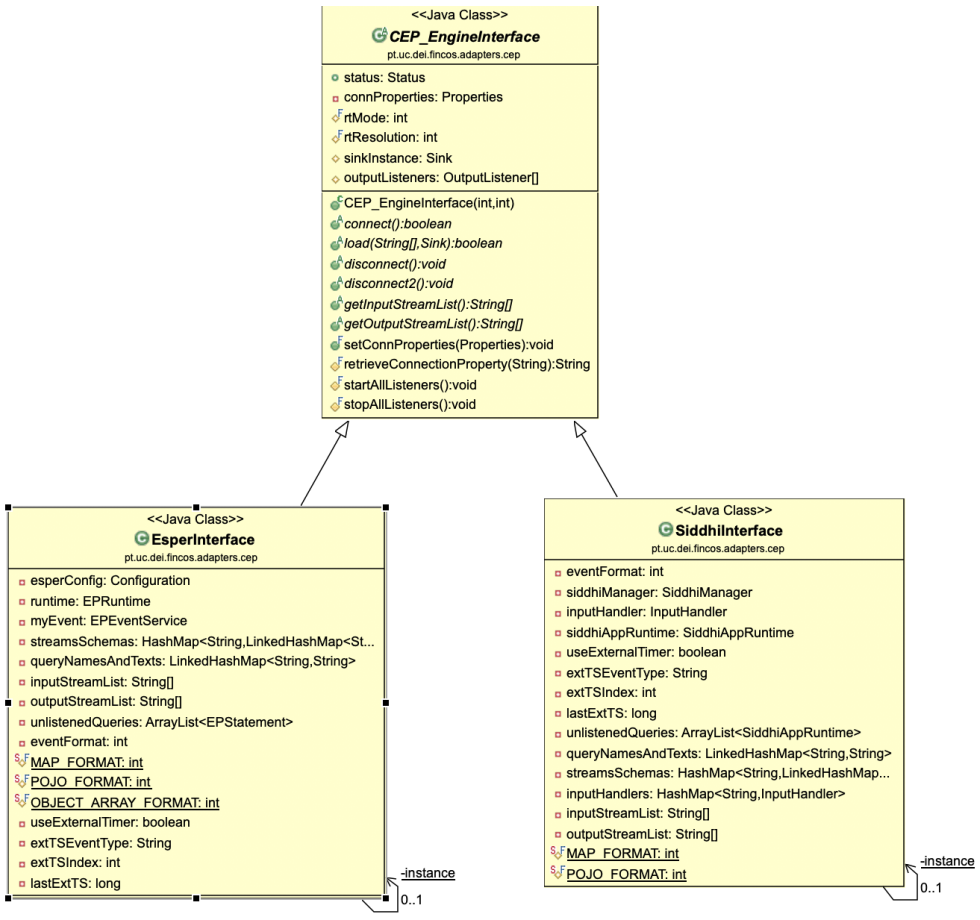


Figura A.8: Diagrama de clases EngineInterface

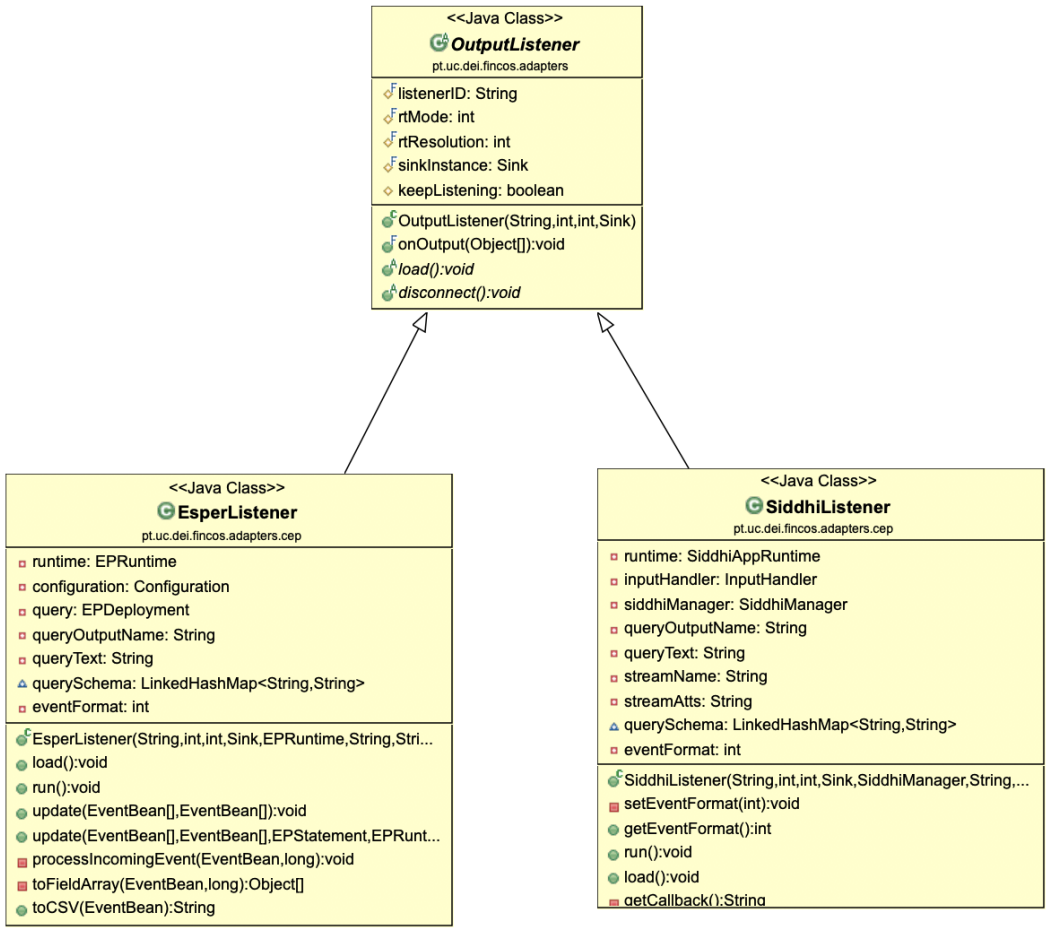


Figura A.9: Diagrama de clases listener

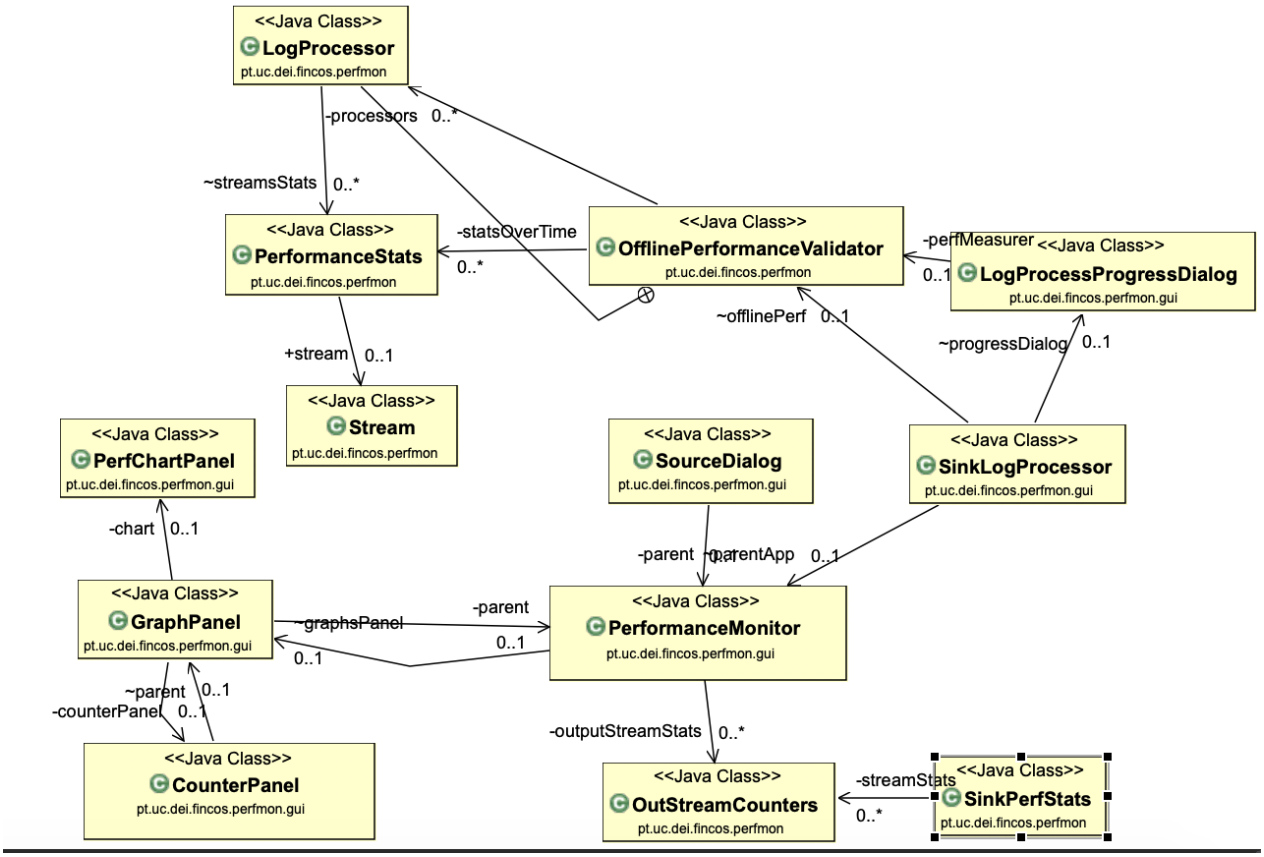


Figura A.10: Diagrama de clases performance monitor